SigMRF Users Guide

Copyright © 2016-2017 Signalogic, Inc.
Revision A2, Apr 2017

Signalogic, Inc.
9617 Wendell @ Skillman
Dallas, TX 75243
Tel: 214-349-5551
Fax: 972-598-0806

http://www.signalogic.com
tech_support@signalogic.com

 Copyright © Signalogic, Inc. 2016-2017

TABLE OF CONTENTS

1 Overview 1

2 SigMRF Architecture 2

2.1 Software Architecture 2
2.1.1 Libraries 3
2.1.2 Background Process 3
2.2 Platform Architecture 4
2.2.1 Packet I/O 4
2.2.2 DPDK 4
2.2.3 coCPU™ Option 5

3 SigMRF Functionality 9

3.1 Packet Flow 9
3.2 Frame Flow 13
3.3 Codecs 14
3.3.1 Voice Codecs 14
3.3.2 Video Codecs 15
3.4 mediaTest Demo, Test, and Measurement Program 16
3.5 APIs 21
3.5.1 Pktlib 21
3.5.2 Voplib 25
3.5.3 DirectCore™ 26
3.5.4 Session Data and Session Command Structs 27

 Copyright © Signalogic, Inc. 2016-2017

Revision History

Created	Rev A1	Nov 2016	SigMRF Users Guide
Modified	Rev A2	Apr 2017	Add pseudocode example for packet and frame processing

 Copyright © Signalogic, Inc. 2016-2017

1 Overview

SigMRF (Signalogic Media Resource Functions) software provides voice and video transcoding, transrating, algorithms, packet processing, statistics, logging, resource management, redundancy, and other media related functionality for the following application areas:

- Telecom infrastructure such as IMS LTE and VoLTE test and simulation equipment
- IT security
- Lawful interception and recording
- Session Border Controllers (SBCs)
- Media Gateways, including extreme capacity systems
- Augmented Reality (AR) systems that require real-time, bidirectional video data flow and processing
- Data analytics and data streaming applications that require voice and video interfaces
- Content delivery

SigMRF is designed to scale from low capacity, relatively simple systems to high capacity, complex systems. When high capacity is required, SigMRF supports coCPU™ hardware that can increase, by a factor of 10x or more, the capacity of cloud servers, as well as small form factor systems (such as mini-ITX), in which size and power consumption constraints can severely limit the number of x86 cores. The coCPU hardware option also supports ATCA blade server systems.

2 SigMRF Architecture

SigMRF software and hardware consists of:

- Libraries (Linux and Windows)
- Background Process (for example, media and packet tasks)
- Test and demo applications (for example, mediaTest)
- Optional coCPU™ hardware

The SigMRF architecture is designed for both flexibility and high performance, supporting multiple data flow and packet I/O options, DPDK, and entry points for user defined applications. User apps can operate and make API calls at several levels, including frame, packet, and session.

2.1 Software Architecture

The SigMRF software architecture includes both libraries and background tasks. Figure 1-1 below shows the SigMRF software architecture.

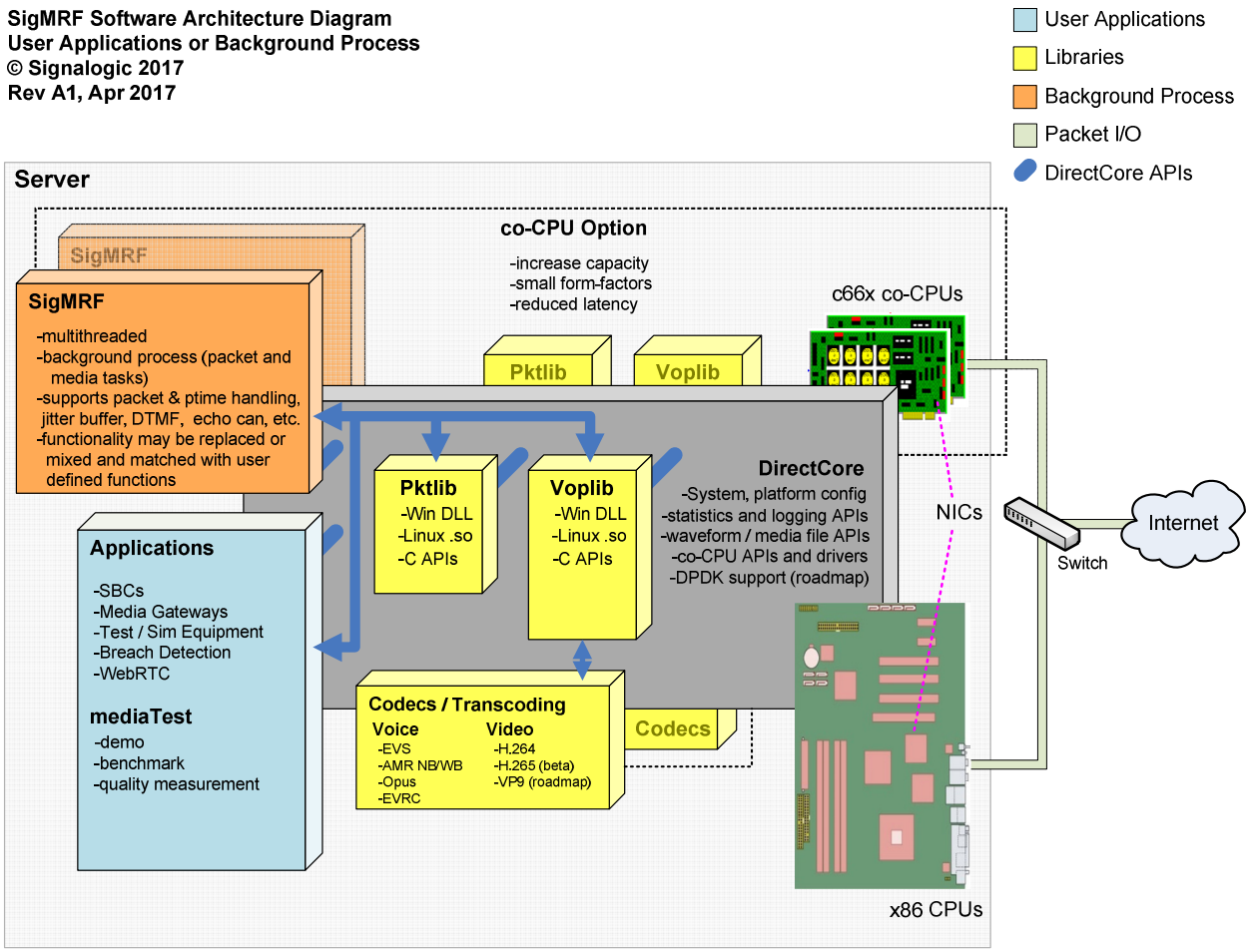

Figure 1-1, SigMRF software architecture and packet I/O

All SigMRF software components are dependent on the DirectCore library for thread-safe access to information about multiple instances, multiple users, and CPU core allocation in cases when explicitly specified. In addition, DirectCore provides an interface layer between software and hardware elements, for example between software and DPDK, software and coCPU™ cores, and software and coCPU NICs.

2.1.1 Libraries

SigMRF provides several libraries, both in Linux (.so [1]) and Windows (DLL [2]) formats, including:

- Pktlib (packet library)
- Voplib (media, or voice / video over packet, library)
- Codec libraries (various voice and video codecs)

Codec libraries are accessed by user applications via Voplib; currently direct access to codec libraries is not supported.

SigMRF also requires the DirectCore set of libraries. DirectCore is a separate product, used by a number of Signalogic application areas, including telecom, image analytics, AI (neural networks), and more. The DirectCore reference guide is located at ftp://ftp.signalogic.com/documentation/DirectCore/DirectCore_Reference_Guide_v3_1_0.pdf.

For Windows, SigMRF shall support only Win10 and above. Some aspects of SigMRF, such as the mediaTest program, used for application reference, demo and test purposes, are expected to require WSL (Windows Subsystem for Linux), which is available only in Win10.

Testing and Development Notes

[1] Pktlib and Voplib Linux shared objects have been tested with Ubuntu v12.04 thru 16.04 (kernel v3.2.0- 45 thru v4.4) and CentOS 7, 7.1 and v6.2. In general, Signalogic software for Linux systems has been tested on tested with Red Hat 9 (Enterprise) 2.6.9- 42 kernel, Ubuntu v12.04 thru 16.04 (kernel v3.2.0- 45 thru v4.4) CentOS 7, 7.1 and v6.2, and Gentoo v2.6.255.

[2] Resources and priority assigned to development of Win10 Pktlib and Voplib DLLs depends on customer needs and business case. Development mostly involves porting existing and tested Linux .so libs to Win10.

2.1.2 Background Process

The SigMRF background process, when needed, runs separately from user applications, and includes:

- Packet task, which runs at a relatively high rate (typically every 1 msec) and handles packet I/O, adding packets to the jitter buffer

- Media task, which runs at a lower rate (typically every 5 msec) and handles packet and payload processing, including transcoding, ptime and transrating, DTMF, and more

Typically, the SigMRF background process comes into play when there is a need for higher performance or user application code must be simplified. Although not all situations require the SigMRF background process to be running, it cannot be separated from SigMRF from a product ordering or support standpoint.

2.2 Platform Architecture

SigMRF supports the following platforms:

- Off-the-shelf rack servers. Typical examples include 1U and 2U form-factors with 8 to 24 Xeon x86 cores
- ATCA server blades. Typical examples include ATCA blades with 4 to 16 x86 or PowerPC cores
- Small form-factor servers. Typical examples include mini-ITX enclosures with 2-4 Atom x86 cores

2.2.1 Packet I/O

SigMRF supports several platform configurations for packet I/O, including:

- Motherboard embedded NICs
- Add-in PCIe card NICs
- Add-in coCPU PCIe card NICs
- Backplane Ethernet interface (ATCA only)

In addition, SigMRF supports packet flow paths between these NIC options including:

- Standard Linux and Windows sockets
- DPDK “fast path” between add-in cards and user applications (Linux only)

2.2.2 DPDK

SigMRF support for DPDK is currently under background development and not documented here. It’s worthwhile to note that existing SigMRF customers have implemented DPDK with coCPU PCIe cards for very high performance server applications.

2.2.3 coCPU™ Option

SigMRF supports coCPU hardware options to (i) substantially increase platform transcoding capacity, (ii) enable transcoding in very small-form factors, and (iii) provide reduced packet latency if needed for certain applications (using NICs located directly on the coCPU hardware, as shown in Figure 2-1 below).

As one example, 64 to 256 coCPU cores can be added to a 1U server; as another example, a 14-slot ATCA rack can host 1920 coCPU cores. These high levels of core density are suitable to address applications constrained in space and power consumption.

From a software architecture perspective, in the diagram in Figure 1-1 above, which uses a partial 3D representation, DirectCore forms a "dividing layer" between x86 CPUs in front of the layer and coCPUs behind the layer. If coCPU hardware is in use, the SigMRF background process, as well as packet and media libraries, exist on both sides of this layer.

The coCPU Users Guide is located at
ftp://ftp.signalogic.com/documentation/Hardware/SigC667x/SigC667x_UserGuide_RevD2.pdf.

coCPU Hardware Options

coCPU hardware options include:

- 32-core and 64-core PCIe cards
- 160 core ATCA blades

coCPU hardware uses Texas Instruments CPUs (8-core c66x devices) which provide the following benefits:

- State-of-the-art CPU architecture, including multicore, L2 cache, DDR3 external mem, onchip PCIe and network I/O, onchip wire-speed packet processor including UDP port filtering, high clock rate SIMD instruction architecture, and extensive onchip DMA
- Run standard C/C++ software without undue modifications, including open source packages of various types
- Build tools generate highly optimized, efficient run-time codes, especially for calculation-intensive applications such as codecs, analytics, neural net convolutions, etc.

Figures 2-1 through 2-4 below show examples of coCPU hardware installed in various off-the-shelf server platforms.

 Copyright © Signalogic, Inc. 2016-2017

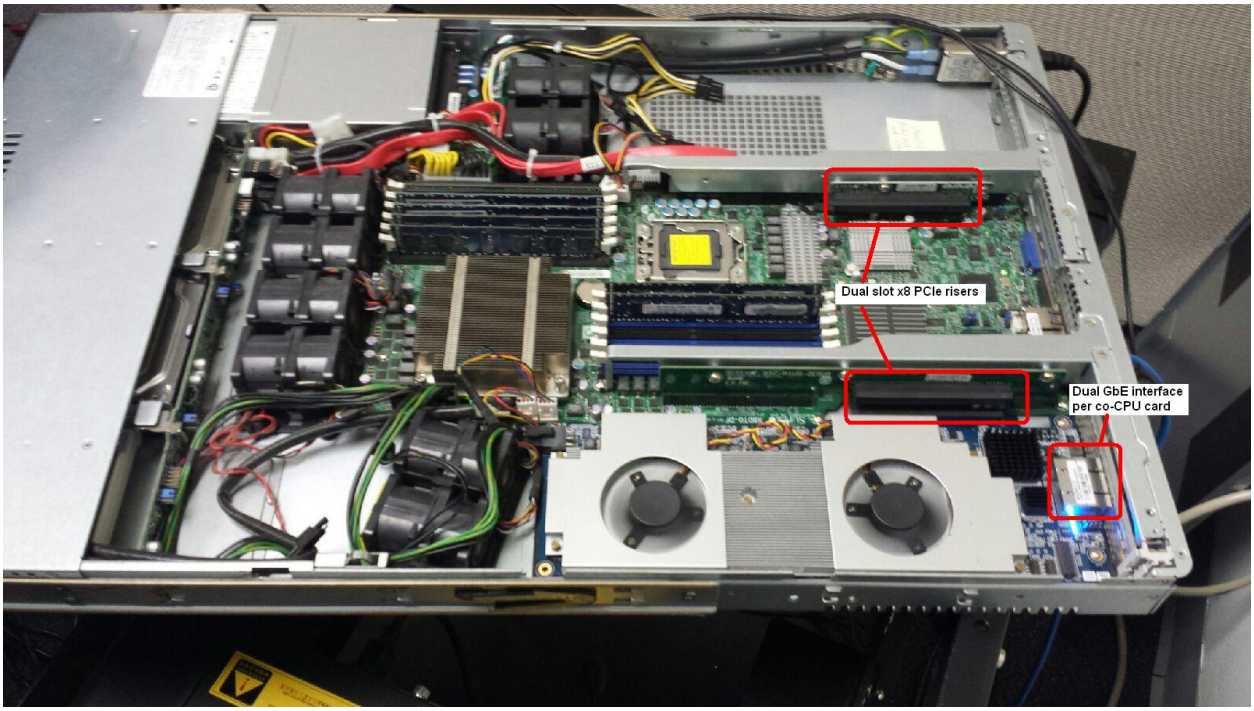

Figure 2-1, coCPU full-length PCIe card installed in a Supermicro 1U server.
Note the dual slot PCIe risers on either side of the motherboard

Figure 2-1 above shows a coCPU full-length PCIe card installed in a low-cost Supermicro 1U server with dual slot PCIe risers on either side of the motherboard. This configuration allows up to 256 coCPU cores to be added to a single 1U server.

Figure 2-2 below shows dual full-length PCIe cards with 128 total coCPU cores, 32 GB DDR3 mem, and 4x GbE interfaces installed in HP DL380 and Dell R720 2U servers.

 Copyright © Signalogic, Inc. 2016-2017

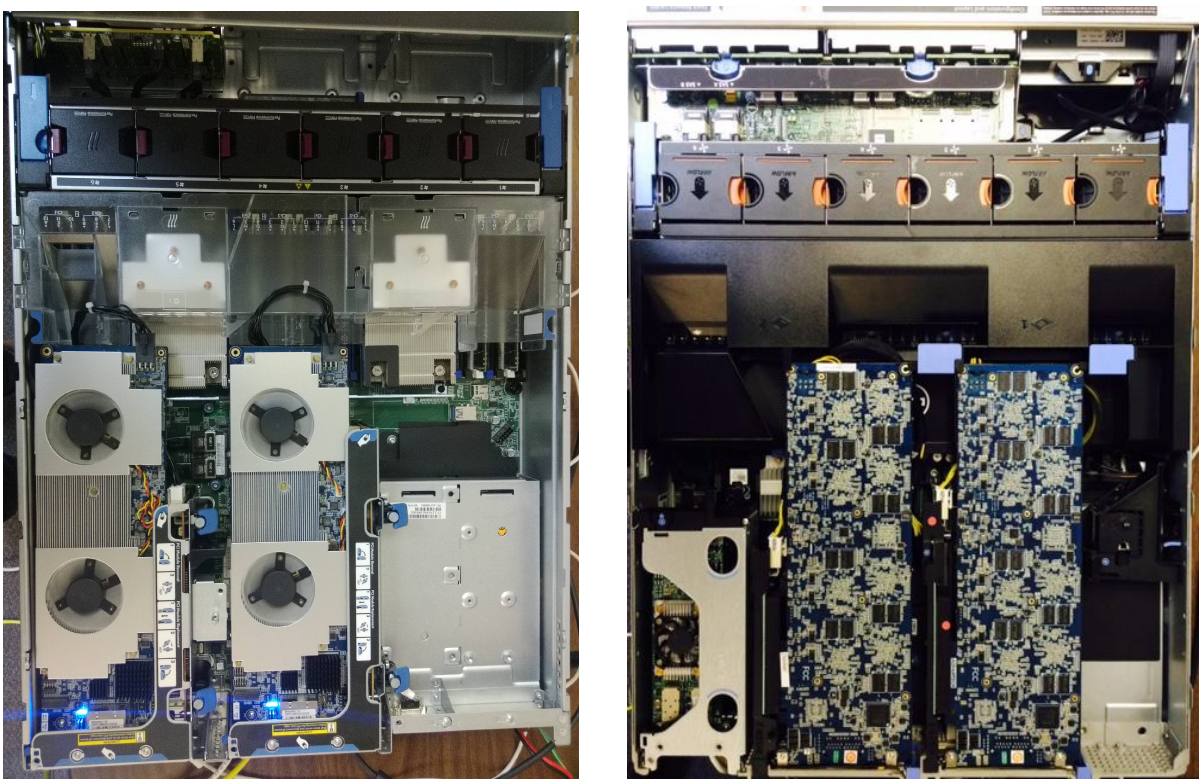

Figure 2-2, coCPU full-length PCIe cards installed in HP DL380 and Dell R720 2U servers

Figure 2-3 below shows a half-length PCIe card with 32 coCPU cores, 8 GB DDR3 mem, and GbE interface installed in a mini-ITX server on a standard PCIe riser. The mini-ITX motherboard has a quad-core Atom CPU, 8 GB memory, and 4x GbE interfaces.

 Copyright © Signalogic, Inc. 2016-2017

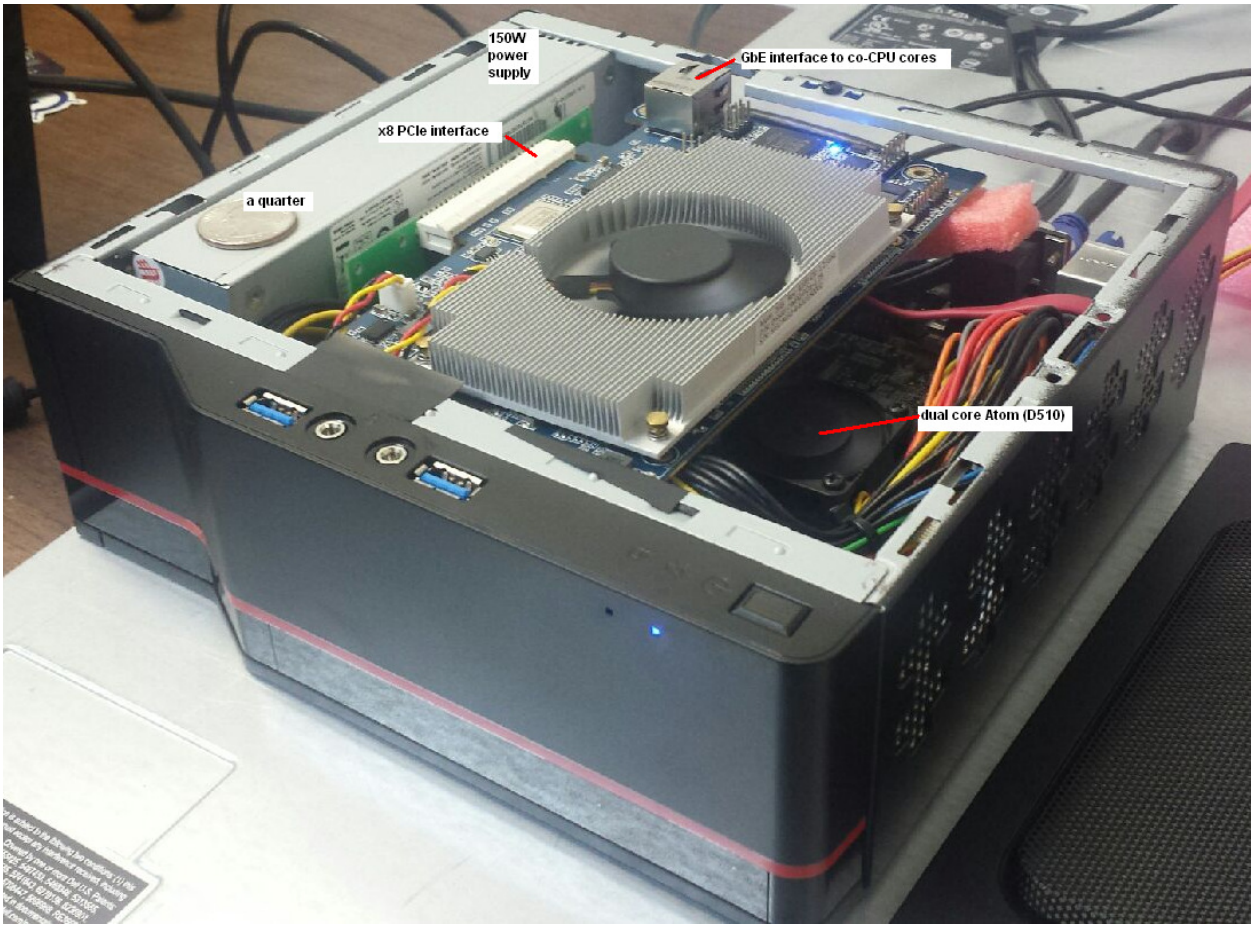

Figure 2-3, coCPU half-length PCIe card installed in a mini-ITX server with enclosure dimensions approx 8" x 9" x 3". Note 100 W power supply at left side of the enclosure

The following links have more information on coCPU hardware and software support:

coCPU HPC Server TI Wiki

Small AI Servers TI Wiki

 Copyright © Signalogic, Inc. 2016-2017

3 SigMRF Functionality

SigMRF software functionality is described in sections below covering data flow, codecs, test programs, and API definitions. Some general notes apply:

- where applicable to mobile and telecom systems, SigMRF software follows 3GPP protocols and standard for media codecs, NFV and C-RAN acceleration, and other areas
- all test and demo programs (i) are multithreaded, (ii) support concurrent multiuser operation, (iii) support KVM + QEMU virtualization, and (iv) support Ubuntu and CentOS systems

3.1 Packet Flow

The SigMRF software architecture supports the following packet flow options:

- User applications receive and sent packets, which are given in Pktlib APIs
- User applications receive and send packets, which are given to the DSSessionTranscode() Pktlib API
- The SigMRF background process invokes Pktlib APIs. The background process takes care of all timing and also calls Voplib APIs

In all cases above, user applications invoke the DSCreateSession() API to specify transcoding and voice algorithm processing details, including jitter buffer, ptime, sampling rate conversion, variable bitrate, DTX and CNG, DTMF, and echo can.. In the first two cases, user applications are responsible for packet I/O and timing. Depending on which Pktlib APIs are called, user applications may also be responsible for some degree of packet parsing and formatting, ranging from all to none.

Figure 3-1 below shows packet and frame data flow in cases where user applications are handling packet I/O and timing, packet parsing and formatting, and not using the SigMRF background process.

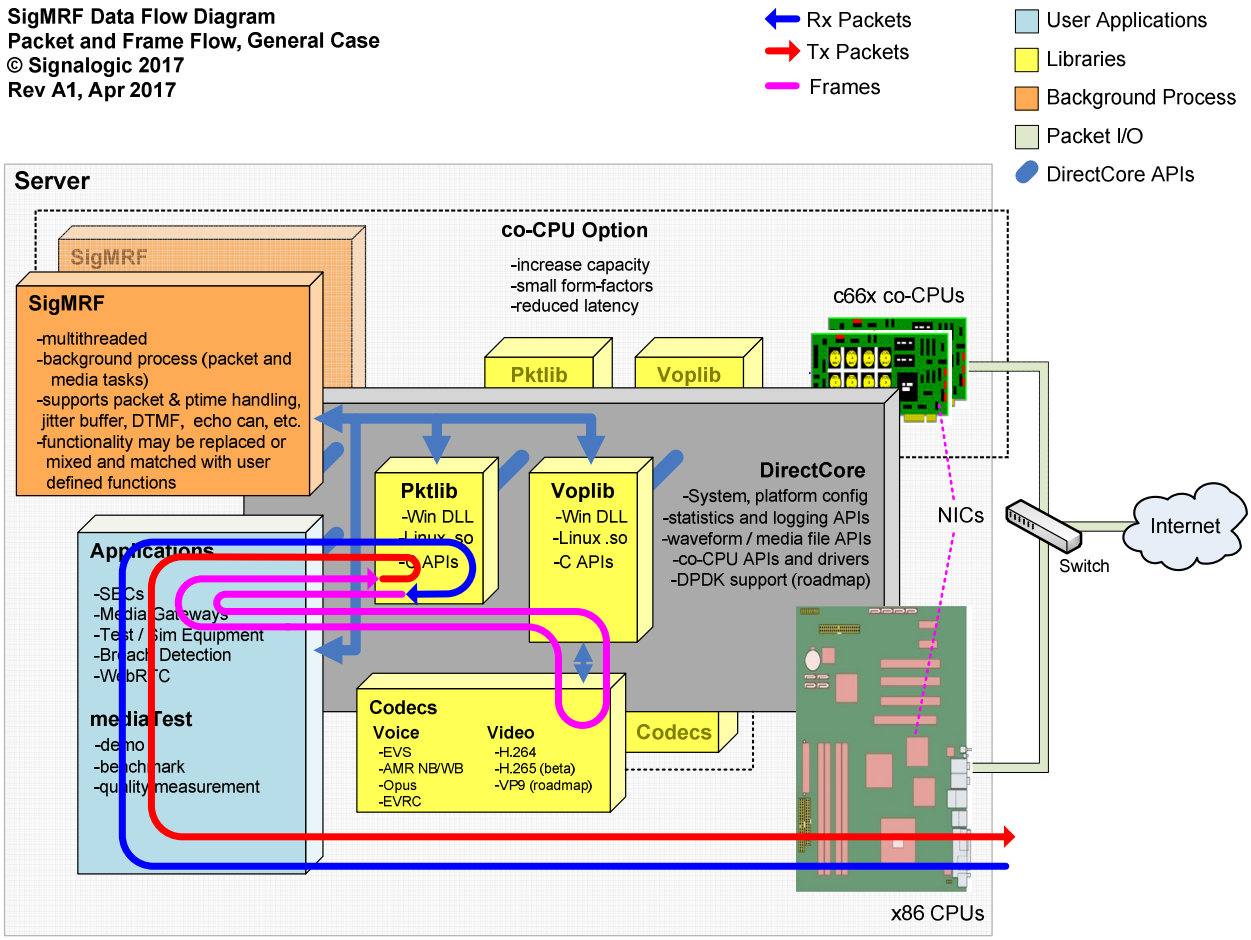

Figure 3-1, User application packet and frame data flow

Below is a pseudocode representation of the processing performed by a user application corresponding to Figure 3-1 above.

```
/* setup code */
#if NONBLOCKING
fcntl(recv_sock_fd, F_SETFL, O_NONBLOCK);
#endif
hDataPlane = DSAssignDataPlane(NULL, "x86", 0, 0, 0, 0, 0);

/* create as many sessions as needed */
[ fill in voice or video attributes and terminfo structs ]
[ fill in session data struct ]
DSSessionCreate(hDataPlane, "none", pSessionData, uFlags)

/* Rx packet processing, performed every 1 msec */
```

```
len = recvfrom(rxpkt)
#if NONBLOCKING
if (len <= 0) continue;
#endif

[ parse packet and get length from packet headers ]

DSBufferPackets(rxpkt)  /* add packet to jitter buffer */

/* media processing and Tx packet handling, performed every 5 msec */
DSGetOrderedPackets(pkt)
hCodec = DSGetPktInfo(pkt, DS_GETPKTINFO_TERM, pTermInfo)
hSession = DSGetPktInfo(pkt, DS_GETPKTINFO_SESSION, pSessionInfo)

[ extract RTP payload ]

DSCodecDecode(hCodec, payload)

[ perform voice processing as needed ]

DSCodecEncode(hCodec, payload)
DSFormatPacket(hSession, pTermInfo, payload, txpkt)
sendto(txpkt)
```

Following are some notes about the above pseudocode:

1. Text marked with [] indicate user defined code.
2. Assumes Linux.
3. The code is divided into three (3) main sections:

    - Setup
    - Rx packet processing (every 1 msec in the example)
    - Tx packet processing (every 5 msec in the example)

    The second two sections would typically run in different threads.
4. The setup section creates a data plane handle and as many sessions as needed. In the example, the data plane handle is created with default parameters (x86 CPUs, no specific core allocation, no clock rate override, etc). Session handles can be stored if needed but cannot be used during multithreaded, asynchronous Rx and Tx packet processing. Instead,

DSGetPktInfo() must be used to retrieve session handles corresponding to (i) received packets and (ii) packets pulled from the jitter buffer prior to voice processing and sending.

The above pseudocode example leaves out many details. For specific, working examples refer to mediaTest source code (see section 3.4 below, mediaTest Demo, Test, and Measurement Program).

Figure 3-2 below shows packet and frame data flow handled by the DSSessionTranscode() API, in cases where user applications are handling packet I/O and timing, but not packet parsing and formatting, and not using the SigMRF background process.

Note that DSSessionTranscode() can also perform packet I/O if a network interface was specified by the session handle in use (session handles are created by the DSSessionCreate() API).

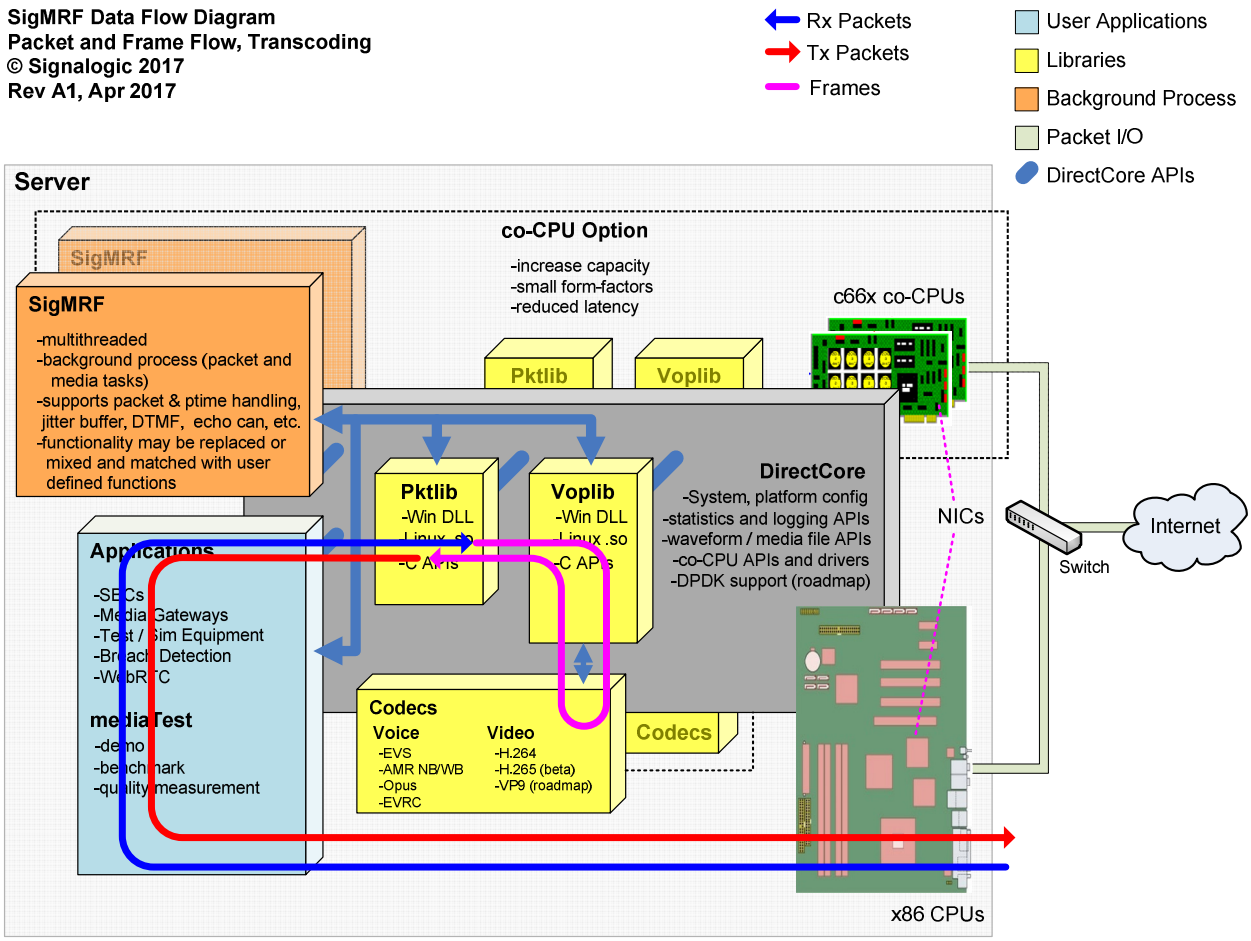

Figure 3-2, User application transcoding packet data flow using DSSessionTranscode()

Figure 3-3 below shows packet I/O, timing, packet parsing and formatting, and packet and frame data flow handled by the SigMRF background process.

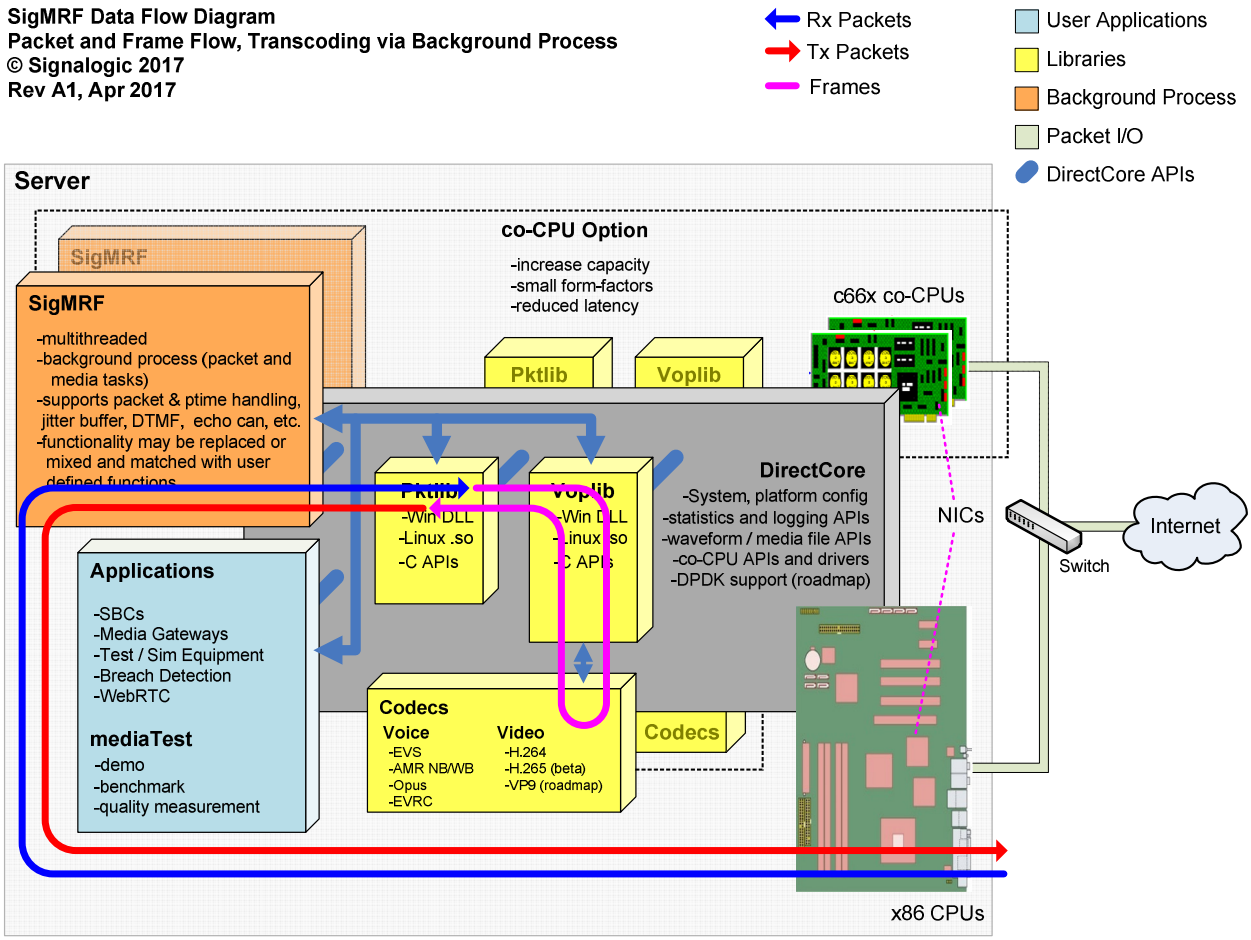

Figure 3-3, Background process transcoding packet and frame data flow

SigMRF background process timing (i.e. rate at which packet and media tasks run) can be configured by Pktlib APIs. The background process runs independently of any user applications.

3.2 Frame Flow

In situations where user applications perform all aspects of packet I/O and timing, packet parsing and formatting, and payload extraction, frames without RTP or other packet headers may be given to Voplib APIs. In this case, the user application is responsible for all aspects of packet flow and processing, including jitter buffer, ptime, sampling rate conversion, aspects of variable bitrate if applicable, CNG and DTX, and DTMF detection and generation.

Figure 3-4 below shows frame data flow.

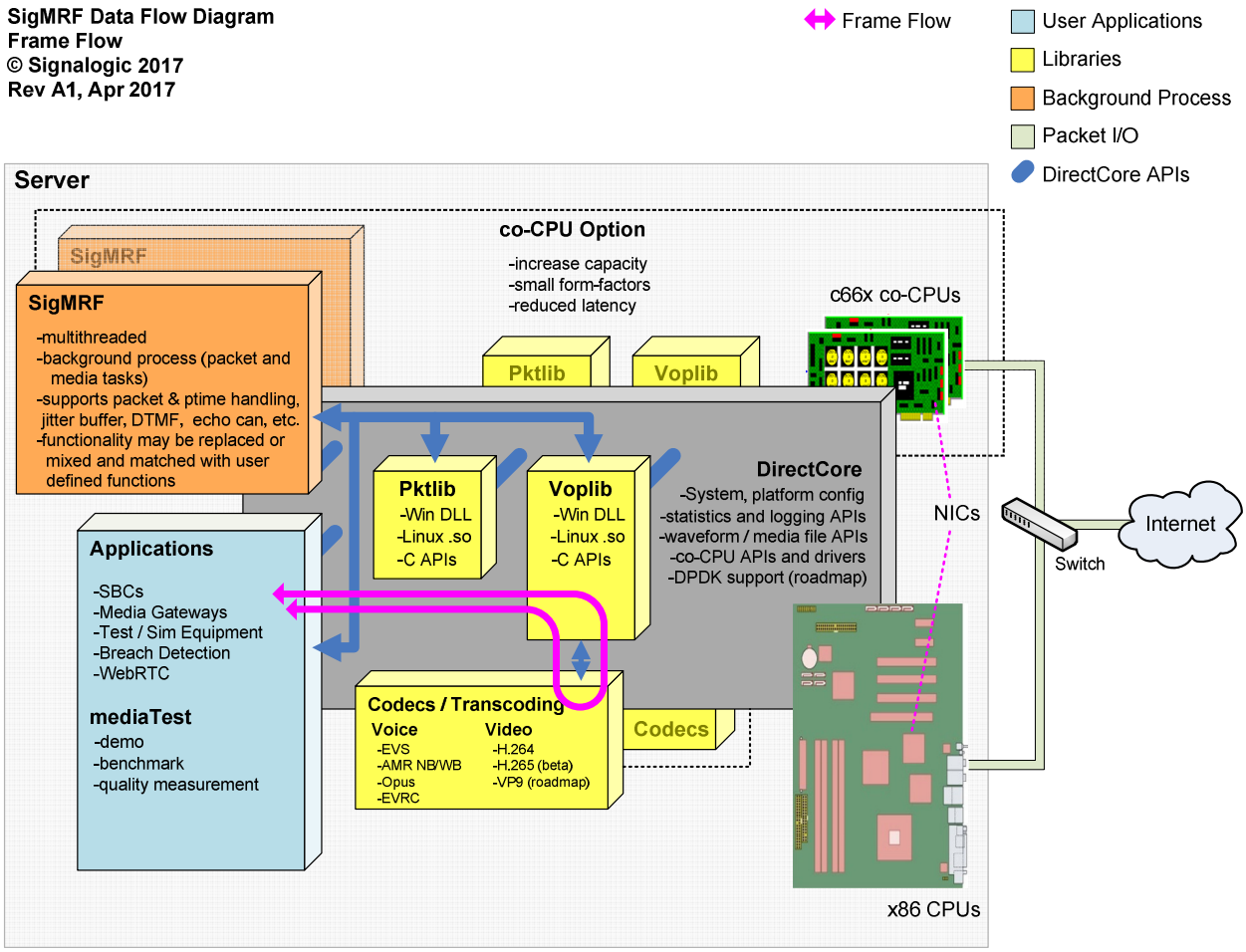

Figure 3-4, Frame data flow

Note that in the case of frame data flow, frame means “compressed bitstream” in a particular codec format for frames being decoded, and “raw audio” (typically in 16-bit PCM format) for frames being encoded.

3.3 Codecs

SigMRF software includes several voice and video codecs, listed in the following sections.

3.3.1 Voice Codecs

SigMRF software includes a number of voice codecs, listed in Table 3-1 below.

Codec	CPU Support [1]	More Information / Comments [2]
EVS	x86, c66x	The x86 version of the EVS codec is based on 3GPP 26.443 source code (floating-point), and coCPU (c66x) is based on 3GPP 26.442, source code (fixed point) Compliance with 3GPP TS 26.445 is given in the HPC EVS Codec Datasheet, located at: ftp://ftp.signalogic.com/data_sheets/Software/EVS_Codec_Datasheet_RevA.pdf The c66x version of EVS is called the HPC EVS codec. More Information is available on the HPC EVS codec web page and the NFV Transcoding TI Wiki: http://signalogic.com/evs_codec.html http://processors.wiki.ti.com/index.php/NFV_Transcoding
G711u, G711A	x86, c66x	http://processors.wiki.ti.com/index.php/NFV_Transcoding c66x version includes Appendix I and II
AMR-WB	c66x	x86 version planned http://processors.wiki.ti.com/index.php/NFV_Transcoding http://processors.wiki.ti.com/index.php/NFV_Transcoding
AMR-NB		
Opus		
EVRC	c66x	x86 version not planned http://processors.wiki.ti.com/index.php/NFV_Transcoding
EVRC-B		
iLBC		
G726		
G729		

Table 3-1, SigMRF voice codecs

Notes

[1] CPU support is updated continuously; current status available upon request.

[2] The links provided contain information about capacity, codec version, CPU and memory usage, and more.

3.3.2 Video Codecs

SigMRF software includes a number of video codecs, listed in Table 3-2 below.

Codec	CPU Support [1]	More Information / Comments [2]
H.264	c66x	http://processors.wiki.ti.com/index.php/NFV_Transcoding http://www.signalogic.com/index.pl?page=dpdk_media_solution Network I/O streaming input and YUV file input supported
H.265	c66x	c66x beta version available; more information available upon request
VP9	c66x	c66x version planned

Table 3-2, SigMRF video codecs

Notes

[1] CPU support is updated continuously; current status available upon request.

[2] The links provided contain information about capacity, codec version, CPU and memory usage, and more.

3.4 mediaTest Demo, Test, and Measurement Program

mediaTest is a Linux command line program that allows demo, test, and measurement for:

- Transcoding
- Network media packet processing
- Codec performance and audio quality

A secondary, but very important, purpose of mediaTest is to provide user application C/C++ source code examples, including Pktlib and Voplib APIs, and coCPU interface. In addition to API usage, mediaTest source code may be used as a general reference for building user applications, including packet flow, timing, multithreading, and other areas.

The mediaTest command line supports the following operating mode options:

- Codec testing , including encode only, decode only, encode-decode, media waveform file formats including .wav, .tim, .cod, and configuration files specifying codec parameters
- Session testing, including multiple concurrent sessions and session configuration files specifying remote / local IP and UDP info, codec type and flags, ptime, sampling rate, bitrate, VAD/DTX, and more
- SigMRF testing
- coCPU hardware configuration – core lists, clock rate, NIC ports, etc

To support transcoding, codec and session configuration files use an numerical endpoint syntax, and session configuration files use a “term info” (termination) syntax.

For codec and session testing, concurrent mediaTest instances in multiple user windows and/or multiple VMs are supported.

Example mediaTest Command Lines

Example mediaTest command lines are shown below.

x86 Motherboard Network I/O and Packet Processing

```
./mediaTest -cx86 -M0 -Csession_config/frame_test_script_config
```

x86 Motherboard Network I/O and coCPU Packet Processing

```
./mediaTest -f1000 -m0xff -cSIGC66XX-8 -e C66xx_RTAF_SYSBIOS_CCSv54.out -M0 -C
session_config/test_config
```

x86 Codec Testing (Back-to-Back Encode-Decode)

```
./mediaTest -cx86 -i reference_files/stv8c.INP -o test_script_files/stv8c_5900_8kHz_dtx_sig.COD -
C session_config/codec_test_8kHz_5900bps_config

./mediaTest -cx86 -ireference_files/stv16c.INP -otest_script_files/stv16c_9600_16kHz_sig.COD -
Csession_config/codec_test_16kHz_9600bps_config

./mediaTest -cx86 -ireference_files/stv32c.INP -otest_script_files/stv32c_13200_32kHz_sig.COD -
Csession_config/codec_test_32kHz_13200bps_config

./mediaTest -cx86 -ireference_files/stv16c_7200_16kHz_mime_$SUFFIX.COD -
otest_script_files/stv16c_7200_16kHz_sig_decoded.OUT -
Csession_config/codec_test_16kHz_7200bps_config

./mediaTest -cx86 -ireference_files/stv16c_13200_16kHz_mime_$SUFFIX.COD -
otest_script_files/stv16c_13200_16kHz_sig_decoded.OUT -
Csession_config/codec_test_16kHz_13200bps_config
```

x86 Codec Testing with Measurement and Logging

```
./mediaTest -cSIGX86 -itest_files/stv16c.INP -otest_files/c6x16c_j.wav -
Csession_config/codec_test_16kHz_13200bps_config
```

coCPU Codec Testing with Measurement and Logging

```
./mediaTest -f1000 -m0xff -cSIGC66XX-8 -etest.out -itest_files/stv8c.INP -otest_files/c6x8c_j.wav
-Csession_config/codec_test_8kHz_13200bps_config

./mediaTest -f1000 -m0xff -cSIGC66XX-8 -etest.out -itest_files/stv16c.INP -
otest_files/c6x16c_j.wav
```

mediaTest Command Line Entry Notes

1) Operating modes may be given in the command line as shown in Table 3-3 below:

Operating Mode Entry	Comments
-M0	Use host network interface (e.g. motherboard network I/O). Packets are processed using x86 or coCPUs, depending on -c cmd line entry

-M1	Simulate network traffic and process simulated packets using coCPUs
-M2	Use coCPU network interface and coCPUs for processing, user app uses PCIe for control plane only (x86 doesn't process packets)
-M3	Test codec encoder, decoder, or both using low level codec APIs, and input / output using cmd line file I/O entry
-M4	Test codec encoder, decoder or both using frame data and Voplib APIs, and input / output using either config files or cmd line file I/O entry

Table 3-3, mediaTest command line operating mode options

2) Input and output audio waveform and compressed data files may be specified in the command line by entering:

```
-iFilepath
-oFilepath
```

in the command line, where Filepath specifies a full or relative path, including filename, for audio files in .wav, .inp, .tim, .au, or other audio format, and / or compressed bitstream files in .cod format.

3) CPU and/or coCPU usage should be given in the command line as shown in Table 3-4 below:

CPU Entry	Comments
-cSIGC66XX-N	Specifies coCPU processing (using one or more c66x PCIe cards, see coCPU User Guide for more information). N is the number of coCPU cores to reserve (default = 8)
-cx86	Specifies x86 processing
-cSIGX86	Specifies x86 processing with measurement and logging options

Table 3-4, mediaTest command line CPU options

4) The following lib folder configuration commands may be needed for console window (shell) operation, depending on SigMRF software installation parameters:

```
ldconfig /install_path/Signalogic/SIG_LIBS/Voice/EVS_fixed-point/lib

ldconfig /install_path/Signalogic/SIG_LIBS/Voice/EVS_floating-point/lib
```

Example Session Config Files

Example session config file entries are shown below.

```
# Session 1

[ start_of_session_data ]

term1.local_ip=10.0.1.211
term1.local_port=10240
term1.remote_ip=10.0.1.71
term1.remote_port=10240
term1.media_type=voice
term1.codec_type=G711_ULAW
term1.bitrate=64000
term1.ptime=20
```

 Copyright © Signalogic, Inc. 2016-2017

```
term1.rtp_payload_type=0
term1.dtmf_type=NONE
term1.dtmf_payload_type=NONE

term2.local_ip=10.0.1.211
term2.local_port=10242
term2.remote_ip=10.0.1.71
term2.remote_port=10242
term2.media_type=voice
term2.codec_type=G711_ALAW
term2.bitrate=64000
term2.ptime=20
term2.rtp_payload_type=8
term2.dtmf_type=NONE
term2.dtmf_payload_type=NONE

[ end_of_session_data ]

# Session 2

[ start_of_session_data ]

term1.local_ip=10.0.1.211
term1.local_port=10244
term1.remote_ip=10.0.1.71
term1.remote_port=10244
term1.media_type=voice
term1.codec_type=G711_ULAW
term1.bitrate=64000
term1.ptime=20
term1.rtp_payload_type=0
term1.dtmf_type=NONE
term1.dtmf_payload_type=NONE

term2.local_ip=10.0.1.211
term2.local_port=10246
term2.remote_ip=10.0.1.71
term2.remote_port=10246
term2.media_type=voice
term2.codec_type=G711_ALAW
term2.bitrate=64000
term2.ptime=20
term2.rtp_payload_type=8
term2.dtmf_type=NONE
term2.dtmf_payload_type=NONE

[ end_of_session_data ]

# Session 3

[ start_of_session_data ]

term1.local_ip=10.0.1.211
term1.local_port=10240
term1.remote_ip=10.0.1.71
term1.remote_port=10240
term1.media_type=voice
term1.codec_type=EVS
term1.bitrate=13200
term1.ptime=20
term1.rtp_payload_type=127
term1.dtmf_type=NONE
term1.dtmf_payload_type=NONE
term1.evs_sample_rate=16000  # in Hz, WB
term1.evs_header_full=0

term2.local_ip=10.0.1.211
term2.local_port=10242
term2.remote_ip=10.0.1.71
term2.remote_port=10242
term2.media_type=voice
term2.codec_type=G711_ULAW
term2.bitrate=64000
term2.ptime=20
term2.rtp_payload_type=0
term2.dtmf_type=NONE
term2.dtmf_payload_type=NONE

[ end_of_session_data ]
```

 Copyright © Signalogic, Inc. 2016-2017

```
# Session 4

[ start_of_session_data ]

term1.local_ip=10.0.1.211
term1.local_port=10244
term1.remote_ip=10.0.1.71
term1.remote_port=10244
term1.media_type=voice
term1.codec_type=EVS
term1.bitrate=13200
term1.ptime=20
term1.rtp_payload_type=127
term1.dtmf_type=NONE
term1.dtmf_payload_type=NONE
term1.evs_sample_rate=16000
term1.evs_header_full=0

term2.local_ip=10.0.1.211
term2.local_port=10246
term2.remote_ip=10.0.1.71
term2.remote_port=10246
term2.media_type=voice
term2.codec_type=G711_ULAW
term2.bitrate=64000
term2.ptime=20
term2.rtp_payload_type=0
term2.dtmf_type=NONE
term2.dtmf_payload_type=NONE

[ end_of_session_data ]
```

Example Codec Config Files

Example codec config file entries are shown below.

```
# Codec 1

[ start_of_codec_data ]

media_type=voice
codec_type=EVS
bitrate=9600
ptime=20
dtx_enable=0
rf_enable=0
sample_rate=32000 # in Hz SWB
header_full=1
encoder_file=reference_files/stv32c.INP
decoder_file=reference_files/stv32c_9600_32kHz_mime.COD

[ end_of_codec_data ]

# Codec 2

[ start_of_codec_data ]

media_type=voice
codec_type=EVS
bitrate=13200
ptime=20
dtx_enable=0
rf_enable=0
sample_rate=32000 # in Hz, SWB
header_full=1
encoder_file=reference_files/stv32c.INP
decoder_file=reference_files/stv32c_13200_32kHz_mime.COD

[ end_of_codec_data ]
```

mediaTest More Information

More information about mediaTest when used with coCPU hardware is located on the Transcoding Test and Demo TI Wiki.

3.5 APIs

Information for Pktlib, Voplib, and DirectCore APIs is given in sections 3.5.x below. In the case of Pktlib and Voplib APIs, header files are included that show:

- API definitions and syntax
- Struct definitions
- Constants
- Attributes and flags

In the case of DirectCore, mostly high level information is given, with references and links included to more detailed information.

3.5.1 Pktlib

Pktlib (Packet library) APIs may be called by user applications to:

- send and receive packets
- add packets to the jitter buffer
- pull ordered / sequenced packets from the jitter buffer
- format payloads into packets prior to sending
- get information about packets in multithreaded, asynchronous data flow, for example hash a packet's contents and look up its associated session data in order operate on the packet

For jitter buffer related APIs, packet format is controlled by flag and attribute values in the API calls. For example, full packets (IP/UDP/RTP format) may be added to the jitter buffer, but only payloads may be extracted Jitter buffer API packet formats include:

- IP/UDP/RTP
- UDP/RTP
- RTP only (i.e. RTP header plus payload)
- Payload only – applies only to pulling from the jitter buffer

Note that relative to the above list, the format of what is pulled must be equal to or a subset of what was added.

All APIs that accept or process RTP packets conform to RFC 3550 when handling RTP headers and payload data.

Pktlib also includes a DSSessionTranscode() API which allows user applications to make one API call per session for transcoding, and let Pktlib handle all transcoding and voice algorithm processing details, including jitter buffer, ptime, sampling rate conversion, variable bitrate, DTX and CNG, DTMF, and echo can.. In this case, user applications are responsible for packet I/O and timing.

Most Pktlib APIs require a session handle, created by the DSSessionCreate() API, which includes:

- a data plane handle, returned by the DSAssignDataPlane() DirectCore API call
- a network interface name pointer (e.g. “em0”, “eth0”, “p6p1”, etc)
- a pointer to a SESSION_DATA struct, which in turn contains a TERMINATION_INFO struct and voice and video attributes structs (see section 3.5.4 below, Session Data and Session Command Structs)
- a uFlags value, a combination of operating modes, data flow paths, and session attributes

Pktlib APIs are implicitly called by the SigMRF background process, including both packet and media tasks. In this case, the background tasks handle all packet I/O and timing.

The following Pktlib header file excerpt definitions for APIs, attributes, and flags, along with descriptive comments.

```
/*
  $Header: /root/Signalogic/DirectCore/include/pktlib.h

  Description: API for creating and managing network traffic sessions and for sending/receiving
  packets to/from pktlib processing buffers

  Project: DirectCore

  Copyright Signalogic Inc. 2017

*/

#ifndef _PKTLIB_H_
#define _PKTLIB_H_

#include "alias.h"

/* session struct and related definitions */
#include "session_cmd.h"

/* typedefs for various pktlib and voplib handles */

#if defined __MSVC__
 typedef void* HSESSION;  /* session handle */
#elif defined _LINUX_
 /* done in alias.h */
#elif defined _WIN32_WINNT
 /* done in alias.h */
#else
 typedef HGLOBAL HSESSION;  /* session handle */
#endif

/* flags for various pktlib functions */
enum
{
   NO_WAIT     /* don't block, return as soon as possible */
```

```
};

#ifdef __cplusplus
extern "C" {
#endif

/* Session management APIs:

    -DSSessionCreate() - create a session to send and/or receive packets on one of following
     network interfaces:

       -none [default]
       -server motherboard
       -network I/O add-in card
       -coCPU card including network I/O

     and returns a session handle (sessionHandle) for use with other packet flow APIs.

     If no network I/O interface is specified, then during subsequent API calls the session is
     matched to packets using IP and UDP header hashing.  networkIfName examples include:

       -eth0, em1 ...    # motherboard NIC
       -p2p1, p6p1 ...   # add-in card or coCPU card

     One each of following operating modes, data flow paths, and timing values should be
     specified:

       -frames only (no packet headers) [default]
       -packet flow with IP/UDP/RTP header
       -packet flow with UDP/RTP header

       -Linux sockets [default]
       -DPDK queue
       -coCPU queue

       -0 -- none, DSSessionTranscode() or other APIs are called by application code based on a
          user-defined timing.  This should be used with frame mode [default]
       -N -- internal SigMRF timing is used and packet and codec related APIs are explicitly
          called every N msec

     The following attributes may be specified:

       -no jitter buffer [default]
       -use the SigMRF internal jitter buffer
       -use the codec built-in jitter buffer (if there is one given by the codec spec)

     Notes:

       -if no network interface is given, DSFormatPackets, DSBufferPackets, and
          DSGetOrderedPackets APIs may be used but not DSSendPackets and DSRecvPackets
       -if a coCPU card, DPDK cores, or specific Linux cores should be used, in either frame or
          packet flow modes, then DSAssignDataPlane() must be called first to obtain a data
          plane handle (dpHandle).  If DSAssignDataPlane() is not called (i.e. a NULL dpHandle is
          used), then SigMRF will use generic multithreading for session handling, and resource
          management APIs are disabled
       -any call to DSSessionCreate() will create and run the internal SigMRF process (if needed
          based on parameters)

    -DSSessionTranscode() -- transcodes one or more packets based on codecs, ptime, and other
       pSessionData params specified in DSSessionCreate().  Notes:

      -calls DSCodecEncode() and/or DSCodecDecode()
      -calls DSBufferPackets(), DSGetOrderedPackets() and DSFormatPackets() if a packet mode was
         given in DSSessionCreate()
      -calls DSSendPackets() and DSRecvPackets() if a network interface was given in
        DSSessionCreate()

    -DSSessionDelete() - delete an existing session.  Notes:

       -the last session deleted will stop and remove the internal SigMRF process (if it was
          running)

*/

HSESSION DSSessionCreate(void* dpHandle, char* networkIfName, SESSION_DATA* pSessionData,
unsigned int uFlags);

int DSSessionTranscode(HSESSION sessionHandle, unsigned int uFlags, uint8_t* pkts);

int DSSessionDelete(HSESSION sessionHandle, unsigned int uFlags);
```

```
/* Packet flow and processing APIs:

    -DSRecvPackets() - receive all available packets associated with the session handle.  Network
       I/O interfaces and data flow paths are same as specified for DSSendPacket().  Notes:

      -block until a packet is avaiable - default behavior
      -immediately return whether or not a packet is available
      -if DS_RECV_PKT_ADDTOJITTERBUFFER is given then packets are added to the SigMRF internal
         jitter buffer
      -TODO: add interrupt based operation
      -TODO: add - setup a callback function to be automatically called whenever a packet is
         available

    -DSSendPackets() - send one or more packets using the network I/O interface and data flow
       path associated with the session handle

    -DSFormatPackets() - format one or more packets for sending, but don't send.  Notes:

      -if DS_SEND_PKT_FMT is given then DSFormatPackets() is called
      -if the session handle is NULL, then packets are matched to an existing session using IP
         and UDP info hashing techniques
*/

int DSRecvPackets(HSESSION sessionHandle, unsigned int uFlags, uint8_t* pkts);

int DSSendPackets(HSESSION sessionHandle, unsigned int uFlags, uint8_t* pkts);

int DSFormatPackets(HSESSION sessionHandle, unsigned int uFlags, uint8_t* pkts);

/* Jitter buffer APIs:

    -DSBufferPackets() - add one or more packets to the SigMRF internal jitter buffer.  Notes:

      -should not be used -- or used very carefully -- if DSRecvPackets() is called with
         DS_RECV_PKT_ADDTOJITTERBUFFER
      -if the session handle is NULL, then packets are matched to an existing session using IP
         and UDP info hashing techniques

    -DSGetOrderedPackets() -- get one or more packets from the SigMRF internal jitter buffer.
     Notes:

      -if the session handle is NULL, then packets are matched to an existing session using IP
         and UDP info hashing techniques
*/

int DSBufferPackets(HSESSION sessionHandle, unsigned int uFlags, uint8_t* pkts);

int DSGetOrderedPackets(HSESSION sessionHandle, unsigned int uFlags, uint8_t* pkts);

/* DSCreateSession() definitions */

#define DS_SESSION_MODE_FRAME           1  /* session data in frame format */
#define DS_SESSION_MODE_IPPACKET        2  /* session data in IP/UDP/RTP packet format */
#define DS_SESSION_MODE_UDPPACKET       3  /* session data in UDP/RTP packet format */

#define DS_SESSION_LINUX_SOCKETS        0x10  /* packet flow path is Linux sockets */
#define DS_SESSION_DPDK_QUEUE           0x20  /* packet flow path is DPDK mem queue */
#define DS_SESSION_COCPU_QUEUE          0x40  /* In frame operating mode, frame flow path is
coCPU card mem queue.  In packet operating mode, packets flow through-CPU card mem queue but are
not processed on the card */

#define DS_SESSION_NO_JITTERBUFFER      0x0
#define DS_SESSION_USE_JITTERBUFFER     0x1000

/* DSRecvPackets() definitions */

#define DS_RECV_PKT_ADDTOJITTERBUFFER   0x1

/* DSSendPackets() definitions */

#define DS_SEND_PKT_FMT                 0x1

#ifdef __cplusplus
}
#endif

#endif   /* _PKTLIB_H_ */
```

3.5.2 Voplib

Voplib (Voice and Video over packet library) APIs may be called by user applications to:

- decode frames of compressed bitstream data for a particular codec (typically formed by extraction from RTP packet payloads), and return frames of raw audio data
- encode frames of raw audio data, and return frames of compressed bitstream data for a particular codec

Raw audio data is typically in 16-bit audio PCM format.

All Voplib APIs require a codec handle, created by the DSCodecCreate() API, which includes:

- a pointer to a TERMINATION_INFO struct, which in turn contains voice and video attributes structs (see section 3.5.4 below, Session Data and Session Command Structs)
- a uFlags value, a combination of operating modes, data flow paths, and codec attributes

Voplib uses a memory model known as XDAIS, an algorithm interoperability standard defined by Texas Instruments, under which all mem allocation occurs at codec creation time and none at run-time (other than local stack inside functions).

The following Voplib header file excerpt gives definitions for APIs, attributes, and flags, along with descriptive comments.

```
/*
  $Header: /root/Signalogic/DirectCore/include/voplib.h

  Description: API for creating and managing audio transcoding sessions

  Project: DirectCore

  Copyright Signalogic Inc. 2015-2017

*/

#ifndef _VOPLIB_H_
#define _VOPLIB_H_

  #include "alias.h"

/* session struct and related definitions */
  #include "session_cmd.h"

/* headers for specific codecs supported - modify as needed, can these be moved to voplib_priv.h?
*/
  #include "Voice/EVS_floating-point/inc/evsenc_sig.h"
  #include "Voice/EVS_floating-point/inc/evsdec_sig.h"

/* Sampling frequency and frame size max values, min ptime */
   #define MAX_PTIME          60       /* in msec */
   #define MIN_PTIME          20       /* in msec */
   #define MAX_FS             48       /* 48 kHz */
   #define MAX_RAW_FRAME      1920     /* MAX_FS*MAX_PTIME*sizeof(short) */
   #define MAX_CODED_FRAME    321      /* EVS: 2560 bits payload size for 128 kbps = 320 bytes +
1 byte ToC */

/* typedefs for various voplib handles */

  #if defined __MSVC__
    typedef void* HCODEC;  /* channel handle */
  #elif defined _LINUX_
    /* done in alias.h */
  #elif defined _WIN32_WINNT
```

```
    /* done in alias.h */
  #else
    typedef HGLOBAL HCODEC;  /* channel handle */
  #endif

/* Operation modes and corresponding APIs:
   -Packet-based transcoding
      -Use pktlib APIs where media type is set to speech in session data
   -Frame-based encoding/decoding/transcoding
      -DSCodecCreate() - setup an encoder and decoder for a codec with associated termination
         info
      -DSCodecEncode()/DSCodecDecode() - encode/decode a frame with the given codec
      -DSCodecTranscode() - transcode between the two given codecs
      -DSCodecDelete() - teardown the encoder and decoder associated with the handle
*/

#ifdef __cplusplus
extern "C" {
#endif

/* codec instance APIs */

  HCODEC DSCodecCreate(TERMINATION_INFO* pTermData, unsigned int uFlags);
  void DSCodecDelete(HCODEC hCodec, unsigned int uFlags);

  int DSCodecEncode(HCODEC hCodec, unsigned int uFlags,
                    uint8_t* inData,
                    uint8_t* outData, uint32_t *out_frameSize);

  int DSCodecDecode(HCODEC hCodec, unsigned int uFlags,
                    uint8_t* inData, uint32_t in_frameSize,
                    uint8_t* outData);

  int DSCodecTranscode(HCODEC hCodecSrc, HCODEC hCodecDst, unsigned int uFlags,
                       uint8_t* inData, uint32_t in_frameSize,
                       uint8_t* outData, uint32_t *out_frameSize);

/* CODEC_INSTANCE accessory functions */

  int DSGetCodecBitRate(HCODEC hCodec);
  int DSGetCodecRawFrameSize(HCODEC hCodec);

/* XDAIS wrapper functions */

  typedef IALG_Handle ALG_Handle;
  ALG_Handle ALG_create(IALG_Fxns *fxns, IALG_Handle p, IALG_Params *params);
  void ALG_delete(ALG_Handle alg);
  char _ALG_allocMemory(IALG_MemRec memTab[], int n);
  void _ALG_freeMemory(IALG_MemRec memTab[], int n);

#ifdef __cplusplus
}
#endif

#endif   /* _VOPLIB_H_ */
```

3.5.3 DirectCore™

DirectCore is a separate software product that provides an underlying set of libraries needed by SigMRF software as well as a number of other Signalogic software products, including video streaming, image analytics, AI (neural networks).

DirectCore provides functionality in two (2) distinct cases:

- for x86 based platforms -- configuration and control of data plane paths, specific x86 core usage, and DPDK

- for coCPU™ hardware (see section 2.2.3 above, coCPU™ Option) – underlying multiuser and multithreading mechanisms for both bare-metal and virtualized system

support and integration of coCPUs, including physical back-end drivers and virtIO front-end drivers

For the coCPU hardware case, more information on DirectCore functionality is located at http://processors.wiki.ti.com/index.php/DirectCore, including:

1. Multiuser Operation
2. TI-RTOS and Bare Metal Support
3. Debug Capabilities
4. Unified Core List
5. Symbol Lookup
6. DirectCore Host and Guest Drivers
7. DirectCore Libraries
8. Software Model
9. Source Code Examples

    Mailbox Create Examples
    Mailbox Query and Read Examples
    CardParams Struct

10. Installing / Configuring VMs

For certain situations involving coCPU hardware, DirectCore can be used stand-alone.

The DirectCore reference guide is located at ftp://ftp.signalogic.com/documentation/DirectCore/DirectCore_Reference_Guide_v3_1_0.pdf.

3.5.4 Session Data and Session Command Structs

Key to the DSSessionCreate() API are the SESSION_DATA, TERMINATION_INFO, and voice and video attributes structs, which are shown in the session.h and session_cmd.h header file excerpts below.

session.h Header File

```
/*
  session.h

  Voice / video session support for c66x, x86, or combined coCPU platforms

    1) Per-channel definitions.  Each channel (stream), has local and remote endpoints
       (or ingress/Rx and egress/Tx endpoints)

    2) Session, termination, and voice / video attributes struct definitions

  Copyright (C) Signalogic, Inc, 2008-2012

    Created for c64x voice platforms

  Copyright (C) Mavenir Systems and Signalogic, Inc, 2013-2015

    Support for c66x coCPU card PCIe and ATCA blade SRIO interfaces
```

```
  Copyright (C) Signalogic, Inc, 2016-2017

    Add APIs to (i) encapsulate c66x PCIe and SRIO interfaces, and (ii) support x86-only or
    combined x86 and c66x server usage.  APIs are consistent between all use cases

*/

#ifndef _SESSION_H_
#define _SESSION_H_

#include <stdint.h>

#define DYNAMIC_JITTER_ENABLE
#ifdef NPLUS1_BUILD
   #define  MAX_REDUNDANCY 13
#elif ONEPLUS1_BUILD
   #define  MAX_REDUNDANCY 1
#else
  #define  MAX_REDUNDANCY 0
#endif

#define USE_BIT8FIELDS

/* internal channel/stream definition.  An array of this structure is declared in call.c
(ChanInfo_Core) */
enum media_type {

  DS_MEDIA_TYPE_VOICE,
  DS_MEDIA_TYPE_VIDEO
};

enum voice_codec_type {

/* Notes:
   1) for G726 and AMR, bit rate is specified using CHANINFO_CORE bitrate element
   2) "G711_WB" is G711.1
*/
  DS_VOICE_CODEC_TYPE_NONE,           /* pass-thru */
  DS_VOICE_CODEC_TYPE_G711_ULAW,      /* 1 */
  DS_VOICE_CODEC_TYPE_G711_ALAW,      /* 2 */
  DS_VOICE_CODEC_TYPE_G711_WB_ULAW,   /* 3 */
  DS_VOICE_CODEC_TYPE_G711_WB_ALAW,   /* 4 */
  DS_VOICE_CODEC_TYPE_G726,           /* 5 */
  DS_VOICE_CODEC_TYPE_G729AB,         /* 6 */
  DS_VOICE_CODEC_TYPE_G723,           /* 7 */
  DS_VOICE_CODEC_TYPE_AMR_NB,         /* 8 */
  DS_VOICE_CODEC_TYPE_AMR_WB,         /* 9 */
  DS_VOICE_CODEC_TYPE_EVRC,           /* 10 */
  DS_VOICE_CODEC_TYPE_ILBC,           /* 11 */
  DS_VOICE_CODEC_TYPE_ISAC,           /* 12 */
  DS_VOICE_CODEC_TYPE_OPUS,           /* 13 */
  DS_VOICE_CODEC_TYPE_EVRCB,          /* 14 */
  DS_VOICE_CODEC_TYPE_GSMFR,          /* 15 */
  DS_VOICE_CODEC_TYPE_GSMHR,          /* 16 */
  DS_VOICE_CODEC_TYPE_GSMEFR,         /* 17 */
  DS_VOICE_CODEC_TYPE_G722,           /* 18 */
  DS_VOICE_CODEC_TYPE_EVRC_NW,        /* 19 */
  DS_VOICE_CODEC_TYPE_CLEARMODE,      /* 20 */
  DS_VOICE_CODEC_TYPE_EVS,            /* 21 */
  DS_VOICE_CODEC_TYPE_INVALID,         /* 22 */
  DS_VOICE_NCODECS = DS_VOICE_CODEC_TYPE_INVALID       /* 22 */
};

enum video_codec_type {

  DS_VIDEO_CODEC_TYPE_MPEG2,
  DS_VIDEO_CODEC_TYPE_H264,
  DS_VIDEO_CODEC_TYPE_VP8,
  DS_VIDEO_CODEC_TYPE_INVALID,
  DS_VIDEO_NCODECS = DS_VIDEO_CODEC_TYPE_INVALID
};

/* default value 0, no dtmf detection or transcoding needed. */
enum dtmf_processing {

    DS_DTMF_NONE = 0x00,       /* equivalent to pass-through */
    DS_DTMF_RTP = 0x01,        /* detect RTP event or outgoing packets are RTP event. */
    DS_DTMF_TONE = 0x02,       /* detect RTP TONE or outgoing packets are RTP TONE */
    DS_DTMF_STRIP = 0x04,      /* remove packet payload for tone */
    DS_DTMF_SIP_INFO = 0x08    /* Reserved */
```

```
};

/* default value 0, no ec */
enum ec_type {

  DS_EC_NONE,         /* NONE */
  DS_EC_TI_LEC,          /* Telogy line ec */
  DS_EC_TI_LEC_ACOUSTIC /* Telinnovations line/acoustic ec */
};

/* VAD: 0 = none, 1= enabled */
#define VOICE_ATTR_FLAG_VAD 0x01

/*
 * comfort noise:  0 = none, 1 = enabled.
 * When enabled, assumes use of VAD (discontinuous transmission)
*/
#define VOICE_ATTR_FLAG_CNG 0x02

/* AMR codec flags */
enum amr_codec_flags
{
    DS_AMR_CHANNELS             = 0x00000007,   /* Possible values = 1 - 6 */
    DS_AMR_OCTET_ALIGN          = 0x00000008,
    DS_AMR_CRC                  = 0x00000010,
    DS_AMR_ROBUST_SORTING       = 0x00000020,
    DS_AMR_INTERLEAVING         = 0x00000040,
    DS_AMR_MODE_CHANGE_PERIOD   = 0x00000080,   /* 0 = 1, 1 = 2 */
    DS_AMR_MODE_CHANGE_CAP      = 0x00000100,   /* 0 - 1, 1 = 2 */
    DS_AMR_MODE_CHANGE_NEIGH    = 0x00000200
};

/*
  EVRC codec flags
  silencesupp is set using VOICE_ATTR_FLAG_VAD
*/
enum evrc_codec_flags
{
    DS_EVRC_FRAME_SIZE      = 0x00000001,   /* 0 = 8 khz, 1 = 16 kHz */
    DS_EVRC_FIXED_RATE      = 0x00000002,   /* 0 = half rate, 1 = full rate */
    DS_EVRC_PACKET_FORMAT   = 0x0000000C,   /* 0 = Interleave/bundled, 1 = header free, 2 =
compact bundled */
    DS_EVRC_BITRATE         = 0x00000070,   /* values from 0 - 7 */
    DS_EVRC_MODE            = 0x00000700,   /* values from 0 - 7 */
    DS_EVRC_MAX_INTERLEAVE  = 0x00007000,   /* values from 0 - 7 */
    DS_EVRC_DTMF            = 0x00010000,   /* values from 0 - 1 */
    DS_EVRC_TTY_MODE        = 0x00060000,   /* values from 0 - 3 or 0 - 1 */
    DS_EVRC_NOISE_SUPP      = 0x00080000,   /* values from 0 - 1 */
    DS_EVRC_POST_FILTER     = 0x00100000    /* values from 0 - 1 */
};

#define DS_EVRC_PACKET_FORMAT_SHIFT     2
#define DS_EVRC_BITRATE_SHIFT           4
#define DS_EVRC_MODE_SHIFT              8
#define DS_EVRC_MAX_INTERLEAVE_SHIFT    12
#define DS_EVRC_TTY_MODE_SHIFT          16

enum evrc_packet_format
{
    DS_EVRC_INTERLEAVE_BUNDLED,
    DS_EVRC_HEADER_FREE,
    DS_EVRC_COMPACT_BUNDLED
};

/*
  OPUS codec flags
  DTX is set using VOICE_ATTR_FLAG_VAD
  if maxaveragebitrate = 0, this value is not specified
*/
enum opus_codec_flags
{
    DS_OPUS_MAX_AVG_BITRATE = 0x00FFFFFF,   /* 6000 - 510000 */
    DS_OPUS_STEREO          = 0x01000000,   /* 0 - mono, 1 stereo */
    DS_OPUS_SPROP_STEREO    = 0x02000000,   /* 0 - mono, 1 stereo */
    DS_OPUS_CBR             = 0x04000000,   /* 0 = variable, 1 = constant bitrate */
    DS_OPUS_FEC             = 0x08000000    /* 0 = FEC disabled, 1 = FEC enabled */
};

#ifdef USE_ATCA_EVS_MODS

#define DS_EVS_BITRATE_SHIFT            2
```

```
#define DS_EVS_PACKET_FORMAT_SHIFT       6
#define DS_EVS_RTCP_APP_ENABLE_SHIFT     7
#define DS_EVS_MAX_REDUNDANCY_SHIFT      8
#define DS_EVS_CMR_SHIFT                 13
#define DS_EVS_CH_SEND_SHIFT             15
#define DS_EVS_CH_RECV_SHIFT             17
#define DS_EVS_CH_AW_RECV_SHIFT          19

enum evs_packet_format
{
  DS_EVS_COMPACT,
  DS_EVS_HEADER_FULL
};

enum evs_rtcp_app_enable
{
 DS_EVS_RTCP_APP_DISABLE,
 DS_EVS_RTCP_APP_DSP,
 DS_EVS_RTCP_APP_HOST
};

enum evs_cmr
{
  DS_EVS_CMR_ZERO,      /* 0 = "0"  all CMR values enabled */
  DS_EVS_CMR_ONE,       /* 1 = "1"  CMR must be present in each packet */
  DS_EVS_CMR_MINUS_ONE  /* 2 = "-1" EVS Primary Mode, CMR byte in RTP header is disabled */
};

enum evs_ch_aw_recv
{
  DS_EVS_CH_AW_RECV_MINUS_ONE = 8, /* 8 = "-1" partial redundancy disabled in receive direction
*/
  DS_EVS_CH_AW_RECV_ZERO      = 0, /* 0 = "0"  partial redundancy not used at session startup */
  DS_EVS_CH_AW_RECV_TWO       = 2, /* 2,3,5,7  partial redundancy used at session startup with
the value provided as the offset */
  DS_EVS_CH_AW_RECV_THREE     = 3,
  DS_EVS_CH_AW_RECV_FIVE      = 5,
  DS_EVS_CH_AW_RECV_SEVEN     = 7
};

enum evs_bit_rate
  {
    DS_EVS_BITRATE_5_9_SC_VBR, /* 0 */
    DS_EVS_BITRATE_7_2,        /* 1 */
    DS_EVS_BITRATE_8_0,        /* 2 */
    DS_EVS_BITRATE_9_6,        /* 3 */
    DS_EVS_BITRATE_13_2,       /* 4 */
    DS_EVS_BITRATE_13_2_CA,    /* 5 */
    DS_EVS_BITRATE_16_4,       /* 6 */
    DS_EVS_BITRATE_24_4,       /* 7 */
    DS_EVS_BITRATE_32,         /* 8 */
    DS_EVS_BITRATE_48,         /* 9 */
    DS_EVS_BITRATE_64,         /* 10 */
    DS_EVS_BITRATE_96,         /* 11 */
    DS_EVS_BITRATE_128         /* 12 */
  };

/*
  EVS codec flags
  silencesupp is set using VOICE_ATTR_FLAG_VAD
*/
enum evs_codec_flags
{
    DS_EVS_SAMPLE_RATE     = 0x00000003,  /* 0 = 8 kHz (NB), 1 = 16 kHz (WB), 2 = 32 kHz (SWB),
3= 48 kHz (FB) */
    DS_EVS_BITRATE         = 0x0000003C,  /* values from 0 - 12, see evs_bit_rate enum */
    DS_EVS_PACKET_FORMAT   = 0x00000040,  /* 0 = Compact, 1 = Header-Full, see evs_packet_format
enum */
    DS_EVS_RTCP_APP_ENABLE = 0x00000180,  /* 0 = Disable, 1 = DSP, 2 = HOST, see
evs_rtcp_app_enable enum */
    DS_EVS_MAX_REDUNDANCY  = 0x00001E00,  /* values from 0 - 11 (Max # Redundant Frames per RTP
packet) */
    DS_EVS_CMR             = 0x00006000,  /* 0 = "0"  all CMR values enabled */
                                          /* 1 = "1"  CMR must be present in each packet */
                                          /* 2 = "-1" EVS Primary Mode, CMR byte in RTP header is
disabled */
                                           /* see evs_cmr enum */
    DS_EVS_CH_SEND         = 0x00018000,  /* Number of audio channels supported in send direction
{1..2}, not present = 1, default = 1 (mono) */
    DS_EVS_CH_RECV         = 0x00060000,  /* Number of audio channels supported in recv direction
{1..2}, not present = 1, default = 1 (mono) */
```

```
    DS_EVS_CH_AW_RECV      = 0x00780000   /* Specifies how channel aware mode is configured or
used in the receive direction {-1, 0, 2, 3, 5, 7} */
                                          /* 8 = "-1" partial redundancy disabled in receive
direction */
                                          /* 0        partial redundancy not used at session
startup */
                                          /* 2,3,5,7  partial redundancy used at session startup
with the value provided as the offset */
                                           /* see evs_ch_aw_recv enum */
};

#else

/*
  EVS codec flags
  silencesupp is set using VOICE_ATTR_FLAG_VAD
*/
#define DS_RF_FEC_OFFSET_SHIFT 13
#define DS_DTX_VALUE_SHIFT 18
enum evs_codec_flags
{
    DS_EVS_SAMPLE_RATE     = 0x00000003,   /* 0 = 8 kHz (NB), 1 = 16 kHz (WB), 2 = 32 kHz (SWB),
3= 48 kHz (FB) */
    DS_EVS_BITRATE         = 0x0000003C,   /* values from 0 - 12 */
    DS_EVS_PACKET_FORMAT   = 0x00000040,   /* 0 = Compact, 1 = Header-Full */
    DS_EVS_RTCP_APP_ENABLE = 0x00000080,   /* 0 = Disable, 1 = Enable */
    DS_EVS_MAX_REDUNDANCY  = 0x00000F00,   /* values from 0 - 11 (Max # Redundant Frames per RTP
packet) */
    DS_RF_FEC_INDICATOR    = 0x00001000,   /* RF FEC Indicator {0, 1} */
    DS_RF_FEC_OFFSET       = 0x0001E000,   /* RF FEC Offset {-1, 0, 2, 3, 5, 7} */
    DS_RF_FEC_OFFSET_SIGN  = 0x00010000,   /* RF FEC Offset sign bit */
    DS_DTX_ENABLE          = 0x00020000,   /* DTX Enable {0, 1} */
    DS_DTX_VALUE           = 0x01FC0000    /* DTX value {0 = adaptive, 3...100}*/
};

#endif  /* ATCA blade system EVS mods */

struct voice_attributes {

#ifdef __BIG_ENDIAN__
  uint32_t flag : 8;              /* see VOICE_ATTR_FLAG_VAD and VOICE_ATTR_FLAG_CNG */
  uint32_t noise_reduction : 8;   /* noise reduction:  0 = none, 1-5 = type */
  uint32_t ec : 8;                /* see ec_type enum */
  uint32_t ec_tail_len : 8;       /* tail length:  0 = not used, otherwise specified in msec */
#else
  uint32_t ec_tail_len : 8;       /* tail length:  0 = not used, otherwise specified in msec */
  uint32_t ec : 8;                /* see ec_type enum */
  uint32_t noise_reduction : 8;   /* noise reduction:  0 = none, 1-5 = type */
  uint32_t flag : 8;              /* see VOICE_ATTR_FLAG_VAD and VOICE_ATTR_FLAG_CNG */
#endif

#ifdef __BIG_ENDIAN__
  uint32_t dtmf_payload_type : 8; /* DTMF payload type to use in remote (egress) RTP header */
  uint32_t dtmf : 8;              /* Refer to dtmf_processing */
  uint32_t rtp_payload_type : 8;  /* RTP payload type to use in remote (egress) RTP header */
  uint32_t ptime : 8;             /* in msec */
#else
  uint32_t ptime : 8;             /* in msec */
  uint32_t rtp_payload_type : 8;  /* RTP payload type to use in remote (egress) RTP header */
  uint32_t dtmf : 8;              /* Refer to dtmf_processing */
  uint32_t dtmf_payload_type : 8; /* DTMF payload type to use in remote (egress) RTP header */
#endif
  union
  {
      struct
      {
          uint32_t codec_flags;     /* See enum amr_codec_flags */
      } amr;
      struct
      {
          uint32_t codec_flags;     /* See enum evrc_codec_flags */
#ifdef __BIG_ENDIAN__
          uint32_t reserved : 8;
          uint32_t hangover : 8;
          uint32_t dtxmin : 8;
          uint32_t dtxmax : 8;
#else
          uint32_t dtxmax : 8;
          uint32_t dtxmin : 8;
          uint32_t hangover : 8;
          uint32_t reserved : 8;
```

```
#endif
      } evrc;
      struct
      {
          uint32_t codec_flags;             /* See enum opus_codec_flags */
#ifdef __BIG_ENDIAN__
          uint32_t sprop_max_capture_rate : 16;  /* 8000 - 48000 */
          uint32_t max_playback_rate : 16;       /* 8000 - 48000 */
#else
          uint32_t max_playback_rate : 16;       /* 8000 - 48000 */
          uint32_t sprop_max_capture_rate : 16;  /* 8000 - 48000 */
#endif
      } opus;

#ifdef USE_ATCA_EVS_MODS
      struct
      {
          uint32_t codec_flags;             /* See evs_codec_flags */
#ifdef __BIG_ENDIAN__
          uint32_t fixed_sid_update_interval : 16;     /* # frame periods (20 ms/frame) b/w fixed
mode SID updates : {3 .. 100, default = 8} */
          uint32_t adaptive_sid_update_interval : 16;  /* # frame periods (20 ms/frame) b/w
adaptive mode SID updates : {8 .. 50, default = 25} */
#else
          uint32_t adaptive_sid_update_interval : 16;  /* # frame periods (20 ms/frame) b/w
adaptive mode SID updates : {8 .. 50, default = 25} */
          uint32_t fixed_sid_update_interval : 16;     /* # frame periods (20 ms/frame) b/w fixed
mode SID updates : {3 .. 100, default = 8} */
#endif
      } evs;
#else
      struct
           {
               uint32_t codec_flags;             /* See evs_codec_flags */
           } evs;

#endif  /* ATCA blade system EVS mods */
  } u;
};

struct video_attributes {

#ifdef __BIG_ENDIAN__
  uint32_t yres : 16;  /* y resolution in pixels */
  uint32_t xres : 16;  /* x resolution in pixels */
  uint32_t reserved : 16;
  uint32_t fps : 16;   /* frames per sec */
#else
  uint32_t xres : 16;  /* x resolution in pixels */
  uint32_t yres : 16;  /* y resolution in pixels */
  uint32_t fps : 16;   /* frames per sec */
  uint32_t reserved : 16;
#endif
};

#define DS_IPV6_ADDR_LEN 16

enum ip_type {

  DS_IPV4,
  DS_IPV6
};

struct ip_addr {

  enum ip_type type;

  union
  {
     uint32_t ipv4;
     uint8_t ipv6[DS_IPV6_ADDR_LEN];
  } u;
};

#ifdef DYNAMIC_JITTER_ENABLE
struct jitter_buffer_config {

#ifdef __BIG_ENDIAN__
/* Delay values are multiples of ptime, eg: 0, 1, 2, 3, etc */
   uint32_t reserved : 8;
   uint32_t min_delay : 8;            /* Minimum allowed delay */
```

```
   uint32_t max_delay : 8;            /* Maximum allowed delay, ultimately limited based on Rx
buffer size (total size of all packets) or heap memory (jitter buffer entries) */
   uint32_t target_delay : 8;         /* Starting delay value, C66x sends alarm when target delay
is consistently not being reached */
   uint32_t decay_coefficient : 16;   /* 'B' coefficient for weighting the current delay value */
   uint32_t attack_coefficient : 16;  /* 'A' coefficient for weighting the new delay value */
#else
   uint32_t target_delay : 8;         /* Starting delay value, C66x sends alarm when target delay
is consistently not being reached */
   uint32_t max_delay : 8;            /* Maximum allowed delay, ultimately limited based on Rx
buffer size (total size of all packets) or heap memory (jitter buffer entries) */
   uint32_t min_delay : 8;            /* Minimum allowed delay */
   uint32_t reserved : 8;
   uint32_t attack_coefficient : 16;  /* 'A' coefficient for weighting the new delay value */
   uint32_t decay_coefficient : 16;   /* 'B' coefficient for weighting the current delay value */
#endif
};
#endif

typedef struct {

  uint32_t term_id;

/* common items for audio and video */

#ifdef USE_BIT8FIELDS

  #ifdef __BIG_ENDIAN__
    uint32_t vqe_processing_interval : 16; /* ms, reserved, currently not used, maybe
configurable per realm */
    /* uint16_t payload_framesize;  // bits */
    uint32_t codec_type : 8;  /* if media_type is VOICE, use voice_codec_type enum, else use
video_codec_type enum */
    uint32_t media_type : 8;  /* see enum media_type definition */
  #else
    uint32_t media_type : 8;  /* see enum media_type definition */
    uint32_t codec_type : 8;  /* if media_type is VOICE, use voice_codec_type enum, else use
video_codec_type enum */
    /* uint16_t payload_framesize;  // bits */
    uint32_t vqe_processing_interval : 16; /* ms, reserved, currently not used, maybe
configurable per realm */
  #endif

#else

   uint32_t vqe_processing_interval; /* ms, reserved, currently not used, maybe configurable per
realm */
   uint32_t media_type;  /* see enum media_type definition */
   uint32_t codec_type;  /* if media_type is VOICE, use voice_codec_type enum, else use
video_codec_type enum */

#endif

  uint32_t bitrate;     /* bps */

  /* Example1: G711    8kHz sampling rate bitrate =  8000 sample/s * 8 bits/sample =  64000 bps
*/
  /* Example2: G711.1 16kHz sampling rate bitrate = 16000 sample/s * 8 bits/sample = 128000 bps
*/
  /* Example3: AMR-NB codec rate 4.75 kbps = 4750 bps */

  struct ip_addr remote_ip;
  struct ip_addr local_ip;
#ifdef __BIG_ENDIAN__
  uint32_t local_port : 16;
  uint32_t remote_port : 16;
#else
  uint32_t remote_port : 16;
  uint32_t local_port : 16;
#endif

#ifdef DYNAMIC_JITTER_ENABLE
  struct jitter_buffer_config jb_config;
#endif

#ifdef USE_ATCA_HOST_TERM_MODS
  uint32_t  pkt_seq_num; /*packet sequence number to be used as base sequence number by dsp for
transcoded packets*/
#endif

  union {
```

```
     struct voice_attributes voice_attr;
     struct video_attributes video_attr;
  } attr;

} TERMINATION_INFO;

#define HOST_TERM_INFO TERMINATION_INFO /* going forward, eliminate reference to "host" under
coCPU architecture, JHB Apr 2017 */

typedef struct {

   uint32_t session_id;
   /* ha_index = 0 (ACTIVE or 1+1 case),  = X = (1..N) (Standby for Active X) */

   uint32_t ha_index;
   TERMINATION_INFO term1;
   TERMINATION_INFO term2;

} SESSION_DATA;

#define HOST_SESSION_DATA SESSION_DATA  /* going forward, eliminate reference to "host" under
coCPU architecture, JHB Apr 2017 */

typedef struct {
   uint32_t term_id;
   uint32_t ssrc;
   uint32_t seq_num;
   uint32_t timestamp;

} HOST_REP_TERM_INFO;

typedef struct {
   uint32_t session_id;

   HOST_REP_TERM_INFO term1;
   HOST_REP_TERM_INFO term2;

} HOST_REP_SESSION_DATA;

#endif /* _SESSION_H_ */
```

session_cmd.h Header File

```
/*
  session_cmd.h

  Session commands and APIs for c66x, x86, or combined platforms

  Copyright (C) Signalogic, Inc, 2008-2012

    Created for c64x voice platforms

  Copyright (C) Mavenir Systems and Signalogic, Inc, 2013-2015

    Support c66x coCPU card PCIe or ATCA blade SRIO interface

  Copyright (C) Signalogic, Inc, 2016-2017

    Add APIs to (i) encapsulate c66x PCIe and SRIO interfaces, and (ii) support x86-only or
    combined x86 and c66x server usage.  APIs are consistent between all use cases

*/


#ifndef _SESSION_CMD_H_
#define _SESSION_CMD_H_

#ifdef __cplusplus
extern "C" {
#endif

#include "session.h"
#include "config.h"
#include "alarms.h"

#include <stdint.h>
```

```
enum cmd_type {

   DS_CMD_CONFIGURATION,                  /* CP -> C66x */
   DS_CMD_HEARTBEAT,                      /* C66x -> CP */
   DS_CMD_OVERLOAD_NOTIFICATION,          /* C66x -> CP */
   DS_CMD_ALARM_INDICATION,               /* C66x -> CP */
   DS_CMD_EVENT_INDICATION,               /* C66x -> CP */
   DS_CMD_CREATE_SESSION,                 /* CP -> C66x */
   DS_CMD_MODIFY_SESSION,                 /* CP -> C66x */
   DS_CMD_DEL_SESSION,                    /* CP -> C66x */
   DS_CMD_CONFIGURATION_ACK,              /* C66x -> CP */
   DS_CMD_CREATE_SESSION_ACK,             /* C66x -> CP */
   DS_CMD_MODIFY_SESSION_ACK,             /* C66x -> CP */
   DS_CMD_DELETE_SESSION_ACK,             /* C66x -> CP */
   DS_CMD_CORE_STATS_QUERY,               /* CP -> C66x */
   DS_CMD_CORE_STATS_RSP,                 /* C66x -> CP */
   DS_CMD_CHANNEL_STATS_QUERY,            /* CP -> C66x */
   DS_CMD_CHANNEL_STATS_RSP,              /* C66x -> CP */
   DS_CMD_DTMF_TONE_GENERATION,           /* CP -> C66x */
   DS_CMD_DTMF_TONE_GENERATION_ACK,       /* C66x -> CP */
   DS_CMD_VAU_NOISE_DETECTION,            /* C66x -> CP */
   DS_CMD_SESSION_TIMEOUT_NOTIFICATION,   /* C66x -> CP */
   DS_CMD_SET_HA_STATE,                   /* CP -> C66x */
   DS_CMD_SET_HA_STATE_ACK,               /* C66x -> CP */
   DS_CMD_REPLICATION,                    /* CP -> C66x for standby */
   DS_CMD_REPLICATION_NOTIFICATION,       /* C66x -> CP for active */
   DS_CMD_CONFIGURE_IP,                   /* CP -> C66x */
   DS_CMD_CONFIGURE_IP_ACK,               /* C66x -> CP */
   DS_CMD_LOG_QUERY,                      /* CP -> C66x */
   DS_CMD_LOG_RSP,                        /* C66x -> CP */
   DS_CMD_GO_ACTIVE,                      /* CP -> C66x */
   DS_CMD_GO_ACTIVE_ACK,                  /* C66x -> CP */
   DS_CMD_TIME_SYNC,                      /* CP -> C66x */
   DS_CMD_TIME_SYNC_ACK,                  /* C66x -> CP */

   DS_CMD_MAX

};

enum ack_type {

   DS_ACK_NONE,
   DS_ACK_OK,
   DS_ACK_UNRECOGNIZED_CMD,
   DS_ACK_UNRECOGNIZED_ID,
   DS_ACK_CMD_INTEGRITY_ERR,
   DS_ACK_SESSION_FULL,
   DS_ACK_CHANNEL_FULL,
   DS_ACK_INVALID_IP_TYPE,
   DS_ACK_CHANNEL_NOT_FOUND,
   DS_ACK_SESSION_NOT_FOUND,
   DS_ACK_DUPLICATE_CHANNEL,
   DS_ACK_UNEXPECTED_COMMAND,
   DS_ACK_INVALID_SESSION_DATA,
   DS_ACK_INVALID_TERM_INFO,
   DS_ACK_CHANNEL_INITIALZATION_FAILED,
   DS_ACK_SESSION_INDEX_FAILURE,
   DS_ACK_HIGH_CPU_LOAD,
   DS_ACK_DUPLICATE_SESSION,
   DS_ACK_INVALID_IP_CONFIG,
#ifndef USE_ATCA_SESSION_CMD_MODS
   DS_ACK_INVALID_TIME_STAMP,                  /* NTP TimeStamp sync */
#endif
   DS_ACK_CHANNEL_INDEX_FAILURE,
#ifdef _USE_N1_REDUNDANCY_
   DS_ACK_INVALID_STANDBY_ID,
#endif
   DS_ACK_CHANNEL_IN_USE
};

enum ha_state
{
   DS_STATE_STANDBY_ONE_PLUS_ONE,
   DS_STATE_ACTIVE,
   DS_STATE_STANDBY_N_PLUS_ONE
};

struct cmd_hdr {

#ifdef __BIG_ENDIAN__
   uint32_t len : 16;      /* data length */
```

 Copyright © Signalogic, Inc. 2016-2017

```
   uint32_t type : 16;
#else
   uint32_t type : 16;
   uint32_t len : 16;      /* data length */
#endif
 /* char data[0];         note that data immediately follows a command header in actual use */
};

/* always sent to core 0 */
struct cmd_configuration {

   uint32_t trans_id;
   GLOBAL_CONFIG gf;
};

struct cmd_create_session {

   uint32_t priv;           /* user application transparent info */
   SESSION_DATA session_data;
};

struct cmd_modify_session {

   uint32_t session_id;

   /* ha_index = 0 (ACTIVE or 1+1 case),  = X = (1..N) (Standby for Active X) */
   uint32_t ha_index;

   TERMINATION_INFO new_term;
};

struct cmd_del_session {

   uint32_t session_id;

   /* ha_index = 0 (ACTIVE or 1+1 case),  = X = (1..N) (Standby for Active X) */
   uint32_t ha_index;
};

struct cmd_configuration_ack {

   uint32_t cause_code;  /* see ack_type enums above */
};

struct cmd_create_session_ack {

   uint32_t cause_code;  /* see ack_type enums above */
};

struct cmd_modify_session_ack {

   uint32_t cause_code;  /* see ack_type enums above */
};

struct cmd_del_session_ack {

   uint32_t cause_code;  /* see ack_type enums above */
#ifdef __BIG_ENDIAN__
   uint8_t reserved;
   union
   {
      uint8_t audio;
      uint8_t video;
   } codec_type[2];
   uint8_t _media_type;  /* codec type are used for MIPs adjustment */
#else
   uint8_t _media_type;  /* codec type are used for MIPs adjustment */
   union
   {
      uint8_t audio;
      uint8_t video;
   } codec_type[2];
   uint8_t reserved;
#endif
};

struct cmd_ovld_notification {

  uint32_t Reserved;
   /* TBD */
};
```

```
struct cmd_core_stats_rsp {

#ifdef __BIG_ENDIAN__
   uint32_t cpu_usage_avg : 16;
   uint32_t cpu_usage_peak : 16;  /* usages are in percent */
#ifdef USE_ATCA_SESSION_CMD_MODS
/* no longer used, replaced by separate mem area stats */
   uint32_t mem_usage_avg : 16;
   uint32_t mem_usage_peak : 16;
#endif
   uint32_t onchip_heap_avg : 16;
   uint32_t onchip_heap_peak : 16;
   uint32_t extern_heap_avg : 16;
   uint32_t extern_heap_peak : 16;
   uint32_t extern_heapBuf_avg : 16;
   uint32_t extern_heapBuf_peak : 16;
#else
   uint32_t cpu_usage_peak : 16;  /* usages are in percent */
   uint32_t cpu_usage_avg : 16;
#ifdef USE_ATCA_SESSION_CMD_MODS
/* no longer used, replaced by separate mem area stats */
   uint32_t mem_usage_peak : 16;
   uint32_t mem_usage_avg : 16;
#endif
   uint32_t onchip_heap_peak : 16;
   uint32_t onchip_heap_avg : 16;
   uint32_t extern_heap_peak : 16;
   uint32_t extern_heap_avg : 16;
   uint32_t extern_heapBuf_peak : 16;
   uint32_t extern_heapBuf_avg : 16;
#endif
   uint64_t rx_pkts;
   uint64_t tx_pkts;
   uint64_t drop_pkts;
   uint64_t rx_octs;
   uint64_t tx_octs;
};

struct cmd_channel_stats_req {

   struct ip_addr remote_ip;
   struct ip_addr local_ip;
#ifdef __BIG_ENDIAN__
   uint32_t local_port : 16;
   uint32_t remote_port : 16;
#else
   uint32_t remote_port : 16;
   uint32_t local_port : 16;
#endif
};

struct cmd_channel_stats_rsp {
   uint32_t term_id;
   uint64_t rx_pkts;
   uint64_t tx_pkts;
   uint64_t drop_pkts;
   uint64_t rx_octs;
   uint64_t tx_octs;
   uint32_t rx_inter_arrival_time_min;
   uint32_t rx_inter_arrival_time_max;
   uint32_t rx_avg_jitter;
   uint32_t codec_mode_change_ue_init;
   uint32_t codec_mode_change_uag_init;
};

struct cmd_log_stats_rsp
{
    uint32_t next_log_idx;
    uint32_t log_wrap_flag;
    uint32_t log_buf_address;
    uint32_t avg_polling_time;
};

struct cmd_heartbeat_notification {

   uint32_t time_stamp;
#ifdef __BIG_ENDIAN__
   uint32_t reserved : 16;
   uint32_t cpu_usage_avg : 16;
#else
```

```
   uint32_t cpu_usage_avg : 16;
   uint32_t reserved : 16;
#endif
};

struct cmd_dtmf_tone_generation {

   uint32_t session_id;

   struct ip_addr remote_ip;
   struct ip_addr local_ip;
#ifdef __BIG_ENDIAN__
   uint32_t local_port : 16;
   uint32_t remote_port : 16;
#else
   uint32_t remote_port : 16;
   uint32_t local_port : 16;
#endif

   uint32_t padding_before_tone; /* silence padding before generated tone, duration of silence in
msec */
   uint32_t padding_after_tone; /* silence padding after generated tone, duration of silence in
msec */
   uint32_t tone_timestamp; /* timestamp field of rtp header */
   struct dtmf_event dtmf;
};

struct cmd_dtmf_tone_generation_ack {

   uint32_t cause_code;  /* see ack_type enums above */
};

struct cmd_vau_noise_detection {

   uint32_t session_id;
   uint32_t priv;           /* user application transparent info */
};

struct cmd_session_timeout_notification {

   uint32_t session_id;
   uint32_t priv;           /* user application transparent info */
};

struct cmd_ha_state {
   uint32_t state;      /* see ha_state */
};

struct cmd_ha_state_ack {
   uint32_t cause_code;  /* see ack_type enums above */
};

struct cmd_replication {
   uint32_t buffer_index;      /* either 0 or 1 */
};

struct cmd_replication_notification {

   uint32_t buffer_index;
};

struct cmd_configure_ip {

#ifdef __BIG_ENDIAN__
   uint32_t reserved : 24;
   uint32_t flag : 8;  /* bit 0  = (SET/UNSET IP, 0 = unset, 1 = set),
                          bit 1  = (PHYSICAL IP SELECT, 0 = invalid, 1 = valid),
                          bit 2  = (VIRTUAL IP SELECT, 0 = invalid, 1 = valid) */
#else
   uint32_t flag : 8;  /* bit 0  = (SET/UNSET IP, 0 = unset, 1 = set),
                          bit 1  = (PHYSICAL IP SELECT, 0 = invalid, 1 = valid),
                          bit 2  = (VIRTUAL IP SELECT, 0 = invalid, 1 = valid) */
   uint32_t reserved : 24;
#endif
   struct ip_addr physical_ip;
   struct ip_addr virtual_ip;
   struct ip_addr subnet_mask; /* field only valid for virtual ip */
   struct ip_addr gateway;     /* field only valid for virtual ip */
};

struct cmd_configure_ip_ack {
```

 Copyright © Signalogic, Inc. 2016-2017

```
   uint32_t cause_code;
};


struct cmd_go_active {

   /* ha_index = X = (1..N) (Become Active X) */
   uint32_t ha_index;
};

struct cmd_go_active_ack {

   uint32_t cause_code;
};

struct cmd_NTP_time_sync {

   unsigned long long NTP;
   /* uint32_t  seconds;           // seconds since epoch */
   /* uint32_t  second_fraction;   // seconds fractional value */
   uint64_t  TSC;                /*  TSC value read by user app as close as possible to NTP
timestamp time */
};

struct cmd_NTP_time_sync_ack {

   uint32_t cause_code;
};

#ifdef __cplusplus
}
#endif

#endif  /* _SESSION_CMD_H */
```